UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

TÍTULO: Sistemas Colaborativos mediante el Desarrollo de Software Dirigido por Modelos

AUTORES: Damián Icelo Amado

DIRECTOR: Luis Mariano Bibbó

CODIRECTOR: -

ASESOR PROFESIONAL: -

CARRERA: Licenciatura en Sistemas

Resumen

En la actualidad, los sistemas colaborativos se han convertido en el foco de atención en ámbitos corporativos y educacionales, incluso también en otras áreas como la industria de videojuegos o científicas. Cuando hablamos de ellos, nos referimos a programas con varios usuarios en línea, que colaboran y se coordinan entre sí, para alcanzar un objetivo.

En este trabajo se presenta el desarrollo de una herramienta de software, elaborada con la Arquitectura Dirigida por Modelos (MDA), con el fin de demostrar los beneficios que aporta la metodología y el funcionamiento apropiado de un metamodelo basado en Sistemas Colaborativos. El desarrollo parte de un metamodelo que nos permite generar modelos de Sistemas Colaborativos, para luego, a través de la Arquitectura Dirigida por Modelos, lograr obtener una herramienta, que posibilite a un desarrollador, generar una aplicación web parcial de su modelo. La aplicación web resultante contiene una estructura de sincronización, que se realiza con las tecnologías modernas, y que permite comportarse como un Sistema Colaborativo. De esta forma, el desarrollador final, que será el encargo de emplear la herramienta, obtiene un Sistema Colaborativo extensible a partir de su propio modelo, garantizando las ventajas de utilizar la Metodología y el correcto funcionamiento del metamodelo de Sistemas Colaborativos.

Palabras Clave

Metodología o Arquitectura Dirigida por Modelos (Model-Driven Architecture, MDA), Metamodelo, Transformación de Modelo (Model Transformation), Sistema Colaborativo (Groupware), Awareness, Workspace, Actividad Colaborativa (Collaborative Activity), Proceso Colaborativo (Collaborative Process), Rol de Colaboración (Collaboration Role), TypeScript, Angular, Mongo, Express, NodeJS, Acceleo, JSON Web Token (JWT), Aplicación Web.

Conclusiones

El resultado final es una herramienta basada en MDA, que permite a un desarrollador lograr un sistema web a través de un modelo propio. El sistema resultante es un programa colaborativo que contiene lo necesario para gestionar y comunicar a los usuarios participantes del mismo, como así también sincronizar los eventos que realizan al interactuar con el sistema.

Por otro lado, se demuestra que el metamodelo de sistema colaborativos cumple con su objetivo, facilitando la creación de modelos y reduciendo el costo de desarrollo de este tipo de sistemas.

Trabajos Realizados

Investigación del uso y ventajas de la Arquitectura Dirigida por Modelos; Investigación de las tecnologías adecuadas para la creación de una herramienta de transformación y el sistema resultante, como Acceleo, Angular, MongoDB, NodeJS, Express; Desarrollo de una herramienta utilizando Acceleo para transformar un modelo de Sistemas Colaborativos en una aplicación web.

Trabajos Futuros

Cubrir la totalidad del metamodelo de Sistemas Colaborativos; Crear nuevos tipos de Awareness; Permitir al desarrollador que utilice la herramienta, gestionar la configuración del sistema destino a través de una interfaz de usuario; Mejorar los estilos del sistema resultante para que el desarrollador pueda modificarlos a su gusto en ejecución, por ejemplo, usando "Drag and Drop".

Fecha de la presentación: 09/2021

Universidad Nacional de La Plata
Facultad de Informática
Laboratorio Lifia

Sistemas Colaborativos mediante el Desarrollo de Software Dirigido por Modelos

Damian Amado

Enviado como requerimiento para el grado de:
Licenciado en Informáticas de la UNLP
31 de agosto de 2021

Agradecimientos

Me gustaría expresar mis agradecimientos a:

- A Luis, por su amabilidad, sus enseñanzas y sus consejos.

- A mi familia, mis padres Susana e Icelo, mis hermanas Ludmila y Corina, y mis sobrinos, por su gran apoyo y ser los ejes de mi vida. En especial a mis padres por estar presentes no solo en esta etapa tan importante, sino en todo momento, ofreciéndome y buscando lo mejor para mi persona.

- A Luciana, la mujer de mi vida, por su amor y paciencia.

- A mis amigos de la vida y a los que me dejó la Facultad, Bruno, Juan, Pablo y Sebastián por sus constantes motivaciones.

Índice general

Índice de cuadros

Índice de figuras

Capítulo 1

Introducción y Motivación

1.1. Introducción

En estos últimos años, los sistemas con múltiples usuarios en línea han crecido de manera exponencial, gracias a la expansión de las nuevas tecnologías y las incansables mejoras que se van realizando año tras año en el hardware, por ejemplo, el crecimiento de los dispositivos móviles. Este tipo de sistemas logran que un usuario ubicado en cualquier parte del mundo pueda conectarse e interactuar con otro en tiempo real. Hoy en día, estos sistemas están presentes en gran parte de las áreas informáticas, desde el sector empresarial u organizacional hasta el ocio.

Los sistemas colaborativos son programas que mantienen un conjunto de usuarios en línea, y permiten la comunicación, la colaboración y la coordinación de dichos usuarios con el objetivo de trabajar sobre tareas que tienen en común. Contienen una interfaz de uso compartido que facilita la interacción entre los usuarios con el fin de lograr la cooperación que se necesita para llegar al objetivo. Y a diferencia de sistemas monousuarios, en donde solo participa un usuario, los sistemas colaborativos almacenan y soportan información compartida, es decir, que administran una base de datos de múltiples accesos, aumentando así la complejidad de este tipo de programas ya que obliga a desarrollar la sincronización adecuada de los datos.

A lo largo de la historia, con el crecimiento de las empresas, la tecnología fue avanzando sobre las áreas de negocios, y el software colaborativo fue surgiendo en base a las necesidades de las empresas. Esto dio lugar a la creación de sistemas colaborativos como, por ejemplo,

repositorios de acceso a archivos, herramientas de comunicación (Chats, Foros, Videollamadas, Wikis, etc..), herramientas de seguimiento de tareas, compartición de recursos en tiempo real, pizarras compartidas, videojuegos, entre otros. Con el avance de las tecnologías, los sistemas colaborativos crecieron en base a los requisitos de las empresas, y es evidente que se vieron beneficiadas por las ventajas que este tipo de programas proporciona, porque les permite agilizar el trabajo y la toma de decisiones, independizándose de las ubicaciones físicas de los miembros que participan.

Por otro lado, los modelos son fundamentales para desarrollar software, ya que fomentan la reutilización de los componentes, y ayudan a agilizar la labor de los participantes que se involucran en el proceso de creación. El Desarrollo de Software dirigido por Modelos (MDD, por sus siglas en inglés, "Model-Driven Development") propone un proceso de desarrollo basado en la realización y la transformación de modelos. Se basa en la abstracción, la automatización y la estandarización. En el proceso de desarrollo de software tradicional, los desarrolladores se encargan de crear un modelo de su problema, por ejemplo, en UML, para luego analizarlo e ir convirtiéndolo a código según sus requerimientos, y así lograr el sistema final. Un cambio en el modelo implica volver a analizar e implementar modificaciones en el código o viceversa. En estos casos, los programadores deberían actualizar el modelo, pero no es algo que se realiza de forma habitual, provocando que el modelo vaya quedando desactualizado. En MDD, el enfoque es totalmente distinto, el concepto principal es el modelo, y a través de lo que llamamos transformaciones, se pretende lograr reducir el tiempo de implementación del código. Cuando hablamos de transformaciones nos referimos a un conjunto de herramientas que se encargan de realizar transiciones entre los distintos modelos que pueden surgir durante el proceso, hasta llegar al código puro de un lenguaje de programación. De esta forma, se obtiene un código automatizado, si bien no es el final, gran parte del desarrollo se logra automatizar. Con este punto de vista, si se debe realizar un cambio en el modelo, solo basta con volver a ejecutar las transformaciones con los nuevos cambios, logrando que el desarrollador se abstraiga de esta parte del proceso, ahorrándole tiempo. Más adelante entraremos en detalle sobre las otras ventajas que aporta esta arquitectura.

Este trabajo se focalizó en cumplir dos objetivos relacionados entre sí. En primera parte, demostrar los beneficios que aporta MDD tanto de forma teórica como práctica. Es decir, investigar, analizar y describir sus conceptos y su utilidad, como así también ponerlos en práctica.

Nuestro segundo objetivo, fue desarrollar una herramienta que nos permita aplicar estos conceptos y al mismo tiempo demostrar que el metamodelo de sistemas colaborativos utilizado cumple con su propósito. El metamodelo mencionado fue creado por Luis Mariano Bibbó para la Facultad de Informática de la Universidad Nacional de La Plata, en las referencias bibliográficas se puede encontrar más información sobre su trabajo [1, 2].

1.2. Motivación

Ya hemos mencionado que los sistemas colaborativos se utilizan en numerosas áreas de la informática. Nuestro principal motivo es proporcionar una herramienta que permita reducir el costo de desarrollo, y ampliar el abanico de posibilidades que puede llegar a tener un programador a la hora de crear un sistema de este tipo. Incluyendo las complejidades que pueden ocurrir cuando se programa un sistema colaborativo.

Otro gran incentivo, son las nuevas tecnologías de desarrollo moderno, como es NodeJS, TypeScript y Angular, basadas en JavaScript. Si bien, con el tiempo siempre se relacionaba y delegaba tareas de poca importancia a este lenguaje, lo cierto es que, en la actualidad, y con sus nuevas extensiones, JavaScript se convirtió en un lenguaje de programación lo suficientemente capaz, tanto así que se compara, al mismo nivel, con otros lenguajes tradicionales como lo son C++, Java o .Net.

La diversidad de los sistemas informáticos va creciendo cada día, y surge la necesidad de lograr más niveles de abstracción y disminuir los tiempos de programación. El Desarrollo Dirigido por Modelos, se enfoca en estos puntos, y se basa en lo que realmente representa al sistema, que son los modelos. Esta metodología, no solo facilita el trabajo, a través de la automatización de código, sino que también apunta a mejorar la Ingeniería de Software, la flexibilidad y el mantenimiento del sistema. Por estas características, el Desarrollo Dirigido por Modelos, es de gran atractivo para lograr el objetivo que realmente se desea.

Por último, queremos que nuestra herramienta no solo sea útil para desarrolladores en sus proyectos personales o privados, sino que también aspiramos a que se pueda aplicar dentro del ámbito educacional. La incorporación del trabajo colaborativo dentro de este espacio podría maximizar la participación de los estudiantes y tener un impacto positivo en el aprendizaje.

1.3. Estructura de la Tesis

Este trabajo está estructurado en los siguientes capítulos:

- **Capítulo 2 - Desarrollo de Software Dirigido por Modelos:** En este capítulo presentaremos la arquitectura, comentando su funcionalidad, su importancia y sus principios. Además, repasaremos las principales ventajas que nos aporta, para luego aplicarlas en la práctica.

- **Capítulo 3 – Sistemas Colaborativos:** En este capítulo mencionaremos la definición de los sistemas colaborativos, sus principales conceptos, los tipos que existen y la actualidad de los mismos.

- **Capítulo 4 - Metamodelo de Sistemas Colaborativos:** En este capítulo explicaremos las características y la estructura del metamodelo de sistemas colaborativos, con el objetivo de integrar los conceptos y facilitar la lectura de los posteriores capítulos, donde se comienza a detallar la herramienta desarrollada.

- **Capítulo 5 - Transformación Conceptual:** En este capítulo presentaremos las transformaciones conceptuales de cada elemento del metamodelo utilizado. Es decir, en que se va a convertir cada elemento desde el punto de vista funcional, luego de aplicar la transformación de la herramienta.

- **Capítulo 6 - Tecnologías:** El objetivo de este capítulo es mencionar las tecnologías que se eligieron para la implementación de la herramienta, y comentar los beneficios que aporta cada una de ellas en el desarrollo.

- **Capítulo 7 - Transformación Técnica:** En este capítulo explicaremos las transformaciones técnicas de los elementos del metamodelo. Esto significa, en qué componentes, de la tecnologías utilizadas, se van a convertir los elementos del modelo.

- **Capítulo 8 - Beneficios de la Herramienta y Conclusión:** En este capítulo listaremos y explicaremos los beneficios que se obtienen al utilizar la herramienta y el metamodelo de sistemas colaborativos. Además, al final, se daremos una conclusión de todo el proceso de desarrollo de este trabajo, y de la experiencia que se obtuvo al llevarlo a cabo.

- **Bibliografía:** Listado de todas las referencias bibliográficas que se utilizaron en la etapa de investigación y desarrollo de este trabajo.
- **Anexo A – Estructura de la Herramienta:** Como adicional, este Anexo explicaremos la estructura de directorios de toda la herramienta, para comprender cada uno de ellos, y tenerlo presente en futuros desarrollos sobre la misma.
- **Anexo B – Ejemplo de Robótica:** Para finalizar, en este Anexo exponemos un ejemplo de robótica creado a partir de la herramienta, y así demostrar que cumple con su objetivo y puede aplicarse en diferentes ámbitos.

Capítulo 2

Desarrollo de Software Dirigido por Modelos

2.1. Introducción

El desarrollo de software está compuesto por un proceso de trabajo, en donde se van atravesando etapas que nos garantizan la calidad del resultado final. Estas etapas y el orden de las mismas están dadas por la Metodología de Desarrollo de Software [3] dentro de la Ingeniería de Software [4]. En resumen, para crear un software se debe seguir un conjunto de pasos, que pueden intercambiarse o ser distintos según la Metodología de Desarrollo que se esté utilizando.

No siempre todos los desarrolladores, que aplican estas metodologías, cumplen con todos los pasos a seguir, haciendo que la calidad se vea afectada. Cada etapa tiene sus beneficios, características y además reducen el costo de la próxima etapa facilitando el trabajo. No entraremos en detalle en cada etapa, porque no es el objetivo de este trabajo, pero si se quiere leer sobre el tema, se puede encontrar más información en las referencias bibliográficas al final del documento. La primera etapa, la etapa de relevamiento y análisis, es la más importante de todo el proceso, y es en donde se realiza la recopilación de los requerimientos de nuestro futuro sistema. Luego de obtener los requisitos deseados, se comienza la planificación y se realizan diferentes bosquejos de toda la información obtenida, los llamamos modelos. Existen diferentes modelos y formas de definirlos, pero todos son modelos conceptuales del sistema y son independientes de la tecnología que se vaya a utilizar en el desarrollo. El objetivo de estos modelos es docu-

mentar y obtener una visualización general de lo que realmente se desea realizar y permitirle a los analistas, diseñadores y programadores discutir sobre el tema. En las siguientes etapas los desarrolladores mantendrán el foco en esos modelos y comenzarán a traducirlos en un lenguaje de programación con la tecnología elegida. A esta interpretación y traducción que realiza el desarrollador se lo conoce como Desarrollo de Software Basado en Modelos [5] (MBD, por sus siglas en inglés, Model Based Development). Este tipo de desarrollo no solo implica la interpretación del modelo, sino que también se aplican conceptos y principios de este proceso, si se quiere leer más en las referencias bibliográficas se cita un artículo muy interesante del tema.

MBD es la forma de desarrollo más cotidiana en el mundo de la programación, pero tiene una gran desventaja, y es que a menudo los modelos tienden a quedar obsoletos, y ocurre porque los requerimientos, que van surgiendo, se van aplicando solamente en el código y no se actualizan los modelos como se deben. Dentro del proceso de desarrollo los modelos son las representaciones conceptuales más importante del sistema. Tener un modelo desactualizado puede provocar que futuros desarrollos perjudiquen la solución del mismo, por ejemplo, si se quiere cambiar de tecnología. Además, todo programador parte de un modelo. Si se incorpora alguien nuevo a un equipo, el punto de partida son los modelos, esto puede provocar que no se adquiera lo que realmente el sistema realiza. Más adelante, en este capítulo, entraremos en detalle sobre estos problemas.

El Desarrollo Dirigido por Modelos [6] (abreviado MDD, por sus siglas en inglés, Model Driven Development) no presenta los inconvenientes mencionados previamente, ya que, en este paradigma de desarrollo, los modelos son los principales protagonistas. Su principal objetivo es separar el diseño de la arquitectura y de las tecnologías, permitiendo que se puedan manejar de una manera independiente. De esta forma el modelo no se ve afectado por los cambios que se produzcan en las tecnologías y en las arquitecturas del sistema. Este modelo independiente de la plataforma se lo conoce como PIM (Platform Independent Model), y puede transformarse en uno o más modelos usando diferentes lenguajes, tanto sean de dominio o de propósito general como Java, Python, etc. Estas transformaciones se realizan a través de diferentes herramientas automatizadas, a lo largo de este trabajo volveremos a mencionar el concepto.

2.2. Importancia de los modelos en MDD

En la sección anterior comentamos que el concepto de modelado para entender el problema y proponer soluciones es una práctica habitual y aceptable dentro de la Ingeniería de Software, mucho antes del surgimiento de MDD. Pero también nombramos que existen algunos problemas, cuando se usan los modelos de esta manera, enunciados a continuación.

- **Los modelos se usan solo como documentación:** Los modelos no funcionan como un artefacto activo que contribuya en el proceso de desarrollo.
- **Existen vacíos entre el modelo y la implementación de los sistemas:** Los cambios en el modelo no se reflejan en el código y los cambios en el código no se reflejan en los modelos, sólo se genera el código de los modelos la primera vez y nunca se actualiza.
- **No hay una adecuada combinación de modelos:** Vistas desconectadas de un sistema (desconexión horizontal) y grupos de modelos desconectados (desconexión vertical).
- **No hay transformación de modelos:** Pocos lenguajes de transformación populares y pocas herramientas que soporten estas transformaciones.

Lo anterior les atribuye a los modelos la fama de una costosa y pesada carga que complica la labor de los participantes en el proceso de desarrollo. MDD rescata la importancia de los modelos como estrategia clave para entender y especificar una solución de software y progresivamente obtener la solución final. Las siguientes son algunas definiciones de modelo dentro de la comunidad de MDD (OMG-MDA, 2003; Kleppe et al., 2003).

- Un modelo es la descripción de un sistema (o, de una parte) en un lenguaje bien definido.
- Un lenguaje bien definido es un lenguaje con una forma definida (sintaxis) y significado (semántica) que sea apropiado para ser interpretado automáticamente por un computador (Kleppe et al., 2003).
- Un modelo se presenta con frecuencia como una combinación de dibujos y de texto (OMG-MDA, 2003).

2.3. Finalidad de los modelos

Para crear sistemas de alta complejidad se deben crear modelos que nos permitan analizar el problema y proponer soluciones para cumplir con nuestro objetivo. A lo largo de la historia, los modelos han sido fundamentales en otras ingenierías tradicionales como, por ejemplo, la ingeniería aeronáutica, civil o industrial. Imagínense construir un puente, que une dos ciudades, sin haber creado un modelo previo, sería absurdo porque cualquier error obligaría a reconstruir el puente por completo, algo completamente inadecuado, que aumente el costo y el tiempo de la construcción. Por esta razón los modelos siempre han sido el punto de partida en cualquier ingeniería, ayudándonos a obtener construcciones más predecibles, fiables y de mejor calidad.

En cada una de las ingenierías pueden existir más de un modelo, y cada uno de ellos pueden realizarse con diferentes lenguajes y a un nivel de abstracción distinto. La Ingeniería de Software no es la excepción, ya que se utilizan lenguajes y notaciones que permiten abstraerse de las plataformas y tecnologías subyacentes, permitiendo una interpretación más conceptual del tema. Cada modelo tendrá un objetivo particular y serán útiles en diferentes momentos del proceso de creación.

En MDD, a lo largo del proceso de nuestra ingeniería, se caracterizan los siguientes tipos de modelos:

- **CIM (Computationally-Independent Model):** Representa los modelos independientes de la computación. Este tipo de modelos surge en la etapa de análisis del negocio y se conciben antes del levantamiento de los requisitos. Su enfoque es hacia el sistema y su entorno, ya que los detalles de la estructura no se especifican en esta etapa. También se lo conoce como modelo de negocio o de dominio y está destinado a usuarios sin conocimientos técnicos, que solo conocen el negocio.
- **PIM (Platform-Independent Model):** Este tipo de modelos representa una visión del sistema independiente de la plataforma, como su nombre lo indica. El objetivo es determinar cómo representar mejor el negocio ignorando los detalles técnicos, como el lenguaje a utilizar. Surgen entre la etapa de análisis y diseño.
- **PSM (Platform-Specific Model):** Se derivan de la categoría anterior, y es en este tipo de modelos donde se empiezan a especificar las tecnologías y plataformas que se

usarán para la construcción del sistema. Pueden existir varios modelos de este tipo para diferentes tecnologías, es decir, que un modelo PIM nos permite crear múltiples modelos PSM. Surgen en la etapa de diseño y codificación.

- **Modelo de la Implementación:** Representa el código final de la solución. Surge en la etapa de codificación y prueba. Es el resultado final de todo el proceso, y se realiza en base a la interpretación del modelo PSM.

En base a los tipos de modelos que van creándose durante el proceso de trabajo en MDD, a la par surgen los participantes que son los encargados de realizar las transformaciones entre los tipos. Entre ellos, están los analistas del negocio, son los expertos del dominio, y se encargan de crear el modelo CIM representando la realidad del tema. Los diseñadores y arquitectos encargados de transformar el CIM a PIM, aportando una propuesta de solución, que progresivamente la concretan (transformaciones de PIM a PIM), hasta llegar a un diseño detallado. Por último, el pasaje de PIM a PSM bajo la responsabilidad de los programadores, experimentados en la tecnología. Ellos deciden la manera más adecuada de la implementación de acuerdo a lo que detalla el modelo (OMG-BP, 2004). En la figura 2.1 se puede observar una representación gráfica del orden secuencial de la creación de los modelos y los niveles de transformación.

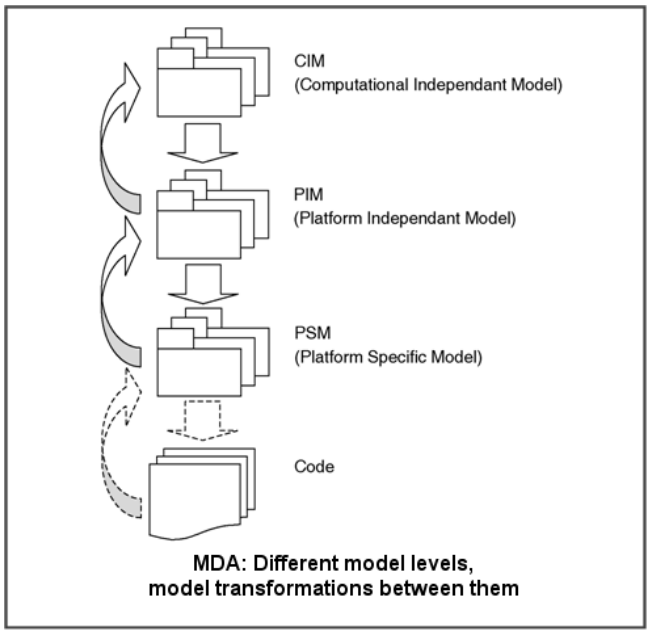

Figura 2.1: Diferentes niveles de modelo y las transformaciones entre ellos.

2.4. Principios de MDD

Los principios en los que se fundamenta MDD son:

- **La diversidad de plataformas y tecnologías:** En la actualidad se oye hablar con frecuencia de los objetos distribuidos, los componentes, los aspectos o los Webs Services, entre otras tantas estrategias tecnológicas en las que no hay mucha interoperabilidad y que tienen tendencia a aumentar.

- **La acelerada evolución tecnológica:** Esto ocasiona que las plataformas se vuelvan obsoletas muy pronto. Surgen, entonces, interrogantes como: ¿Cuál tecnología va a salir mañana? ¿Cuánto va a durar la última versión de una plataforma? ¿Cómo proteger mi inversión? Por consiguiente, nunca se tiene un verdadero estándar en el nivel de sistemas operativos, servidores, plataformas o middleware, lo cual dificulta considerablemente el reúso de los artefactos de software, y más aún en etapas tempranas del proceso. Las estrategias para alcanzar beneficios fundamentales, como productividad, interoperabilidad, portabilidad y facilidad de mantenimiento, se plantean en las ideas del manifiesto MDA (Booch et al., 2004).

- **Representación directa:** Esta estrategia se basa en el principio de abstracción, que hace énfasis en el dominio del problema más que en la tecnología. Los diferentes tipos de modelos mencionados buscan precisar una semántica que claramente separe los aspectos relevantes del problema de las decisiones de tecnología. Esta estrategia parte de la hipótesis de que los impactos considerables en el desarrollo y mantenimiento de una solución de software se dan por cambios en el negocio, más que por la diversidad de plataformas y la evolución tecnológica.

- **Automatización:** La propuesta de MDD fortaleció y dinamizó el papel que las herramientas CASE tienen en el desarrollo de soluciones. Surgen nuevas funcionalidades que deben ser soportadas por las herramientas como el intercambio de modelos, verificación de consistencia, transformación de modelos y manejo de metamodelos, entre otras. Desde la perspectiva de MDD, el papel de las herramientas es esencial para apoyar de forma consistente y sistemática el proceso de desarrollo visto como un proceso de transformación de modelos.

- **Estándares abiertos:** El uso de estándares se ha constituido en el medio que ha posibilitado el reto de integrar herramientas robustas de apoyo al desarrollo. Por ejemplo, los estándares como UML deben expresarse en XML, de esta forma resultan de gran utilidad en mecanismos de transformación de modelos que utilizan otros estándares como XSLT6.

2.5. Metamodelo

Para poder construir un modelo necesitamos conocer lo que ciertamente queremos crear, y distinguir cual es el objetivo de cada uno de sus componentes. Luego, debemos buscar algo que nos permita generar los componentes que realmente representan. Es decir, buscar un mecanismo que nos otorgue de forma genérica la representación de nuestros componentes. Estos mecanismos son los metamodelos. Son modelos, en un nivel superior de abstracción, que nos permiten generar otros modelos relacionados al mismo contexto. Técnicamente, un metamodelo especifica los conceptos de un lenguaje, las relaciones entre ellos, y las reglas estructurales y semánticas que restringen los componentes de los modelos válidos. Cada modelo se escribe en el lenguaje que indica su metamodelo, esto podría verse como que cada modelo es la creación de una instancia de su metamodelo. Por ejemplo, los conceptos genéricos de paquete, clase, atributo y operaciones aparecen en el metamodelo UML. Los conceptos de tabla, columna y claves son parte de SQL.

La OMG define 4 niveles para los modelos:

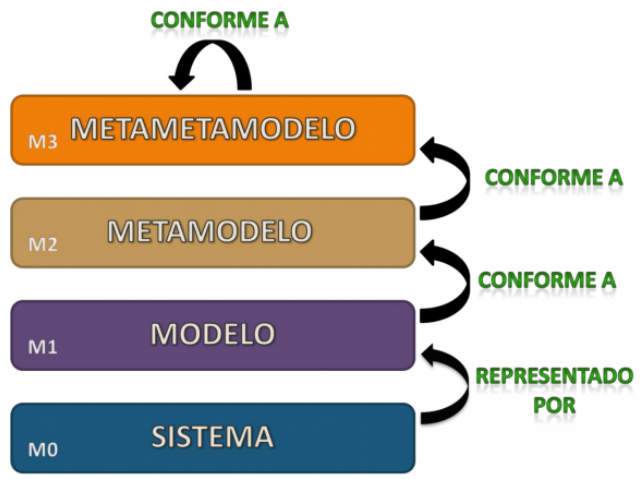

Figura 2.2: Niveles de modelos según OMG.

Un modelo es creado conforme a su metamodelo. A su vez, un metamodelo está escrito en un lenguaje definido por su meta metamodelo. El proceso de abstracción de modelos acaba cuando se llega al nivel superior de meta metamodelo, porque los meta metamodelos son conformes a ellos mismos.

En nuestro caso, usaremos un metamodelo de sistemas colaborativos, que nos permitirá representar, de forma genérica, los elementos usuales que se pueden modelar para este tipo de sistemas. Como lo son, las actividades, los procesos, los roles, los awareness y los eventos.

Capítulo 3

Sistemas Colaborativos

3.1. Introducción

A lo largo de la historia, a través de la evolución de las tecnologías, las computadoras fueron aumentando su velocidad y reduciendo el tamaño de sus componentes físicos. Esto otorgó la posibilidad de que las computadoras se empiecen a comercializar y utilizar para cuestiones personales o empresariales, y no solo en laboratorios o universidades. Esto provocó una importante revolución dentro de los años 60s, en donde cualquier empresa podía adquirir y emplear computadoras para cumplir con sus objetivos. Más adelante, la misma evolución, permitió compartir los recursos de las computadoras, es decir el propio hardware, como la memoria, los discos de almacenamiento y el procesador. Pero no existía la posibilidad de poder compartir la información, por limitaciones como la distancia física entre oficinas y los diferentes sistemas operativos. Luego, la llegada de Internet logró romper estos límites y permitir compartir información, pero seguía existiendo la necesidad de realizar actividades colaborativas. A partir de aquí, se empezó a buscar nuevas tecnologías y mecanismos que permitan satisfacer esta necesidad por parte de las empresas. A lo largo del tiempo y hasta la actualidad, fueron surgiendo, dando la posibilidad de crear este tipo de software, en donde los usuarios colaboran entre sí para cumplir sus objetivos.

3.2. Conceptos de los Sistemas Colaborativos

Como hemos mencionado en alguna ocasión dentro de este documento, un sistema colaborativo es un programa o un conjunto de programas que permiten a múltiples usuarios, conectados en línea o de forma concurrente, participar y colaborar para llegar a un objetivo particular. Estos usuarios se encuentran en diversas estaciones de trabajo, incluso en ocasiones fuera de la misma oficina, y se conectan a través de Internet o de una Intranet.

Hay autores que suelen clasificar al software colaborativo en dos tipos: *Groupware* y *Workflow*. En las siguientes secciones, explicaremos la diferencia entre estos dos conceptos, y concentramos nuestra atención en Groupware, ya que nuestro objetivo es obtener este tipo de sistemas, una vez aplicada la transformación de nuestra herramienta.

3.3. Groupware

Las empresas buscan una mayor productividad para hacer frente a la competencia global en donde se enfrentan a diario. Por esta razón, buscan optimizar sus recursos como las computadoras, materia prima o el conocimiento que aporta el recurso humano, y adaptar sus procesos a los cambios que traen las tecnologías o las nuevas demandas de los clientes. Como alternativa, muchas organizaciones o empresas se apoyan en sistemas que ayudan a la comunicación. Pero sin perder de vista que son los mismos usuarios los que deben realizar el trabajo para entender los procesos en curso.

El *Groupware* es un tipo de sistema colaborativo que consiste en ayudar a un grupo de trabajo a cumplir sus actividades de forma remota. A continuación, se presentan definiciones formales.

'Sistemas basados en computadoras que apoyan a grupos de personas que trabajan en una tarea común y que proveen una interfaz para un ambiente compartido'. -David Chaffney.

'Herramientas computacionales especializadas y diseñadas para el uso de grupos de trabajo colaborativos'. -Robert Johansen.

Las características más importantes de un *Groupware* son:

- Proporcionar un contexto o ambiente de colaboración, intentando simular la misma interacción que ocurre en una oficina real, para lograr que el grupo lleve a cabo su trabajo.
- La información centralizada se mantiene en un solo sitio y es compartida por todos los participantes.
- Interacción en línea, mediante distintas formas, por ejemplo, escrita, voz o vídeo.
- No requiere el traslado físico de los participantes. Se mejora el trabajo entre oficinas que no se encuentran físicamente ubicadas en la misma dirección. Esto implica una reducción de costos para las empresas, ya que transportar personal de un lugar a otro es más costoso que instalar una Intranet o comprar la licencia de un software colaborativo.
- Decisiones más rápidas. Un *Groupware* o sistema colaborativo mejora la toma de decisiones ya que no tiene importancia la distancia entre los miembros del equipo.

Comúnmente, a los sistemas colaborativos también se los suele conocer como *Groupware*. A partir de aquí utilizaremos el término indistintamente.

Dentro de los sistemas colaborativos, el tiempo y el espacio hacen que se puedan clasificar este tipo de sistema en sincrónicos o asincrónicos, en base al tiempo, o si se encuentran de forma centralizada o distribuida, en base al espacio. Un *Groupware* sincrónico (aplicaciones en tiempo real) son: pizarras compartidas, aplicaciones de videollamadas o teleconferencias, chats y sistemas de toma de decisiones. Algunos ejemplos de *Groupware* asincrónicos son: sistemas de manejo de e-mails, calendarios y sistemas de colaboración de escrituras, entre otros. A continuación, se muestran ejemplos de *Groupware's*.

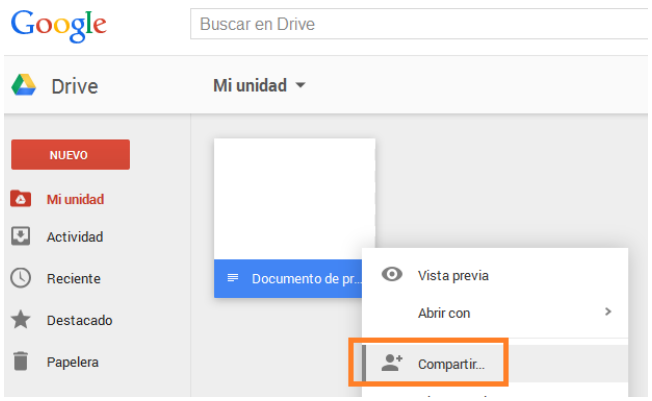

Figura 3.1: Google Drive, Herramienta para compartir y editar documentos en línea.

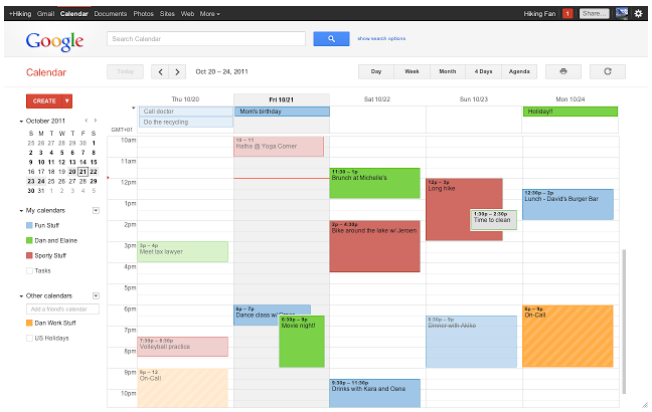

Figura 3.2: Google Calendar, calendario electrónico que permite a sus contactos invitar y sincronizar eventos.

Figura 3.3: Zoom Vídeo, videollamadas y reuniones virtuales.

3.4. Workflow

Como hemos mencionado anteriormente, nuestra atención en este trabajo, son los *Groupware's*, por este motivo en esta sección comentaremos muy brevemente los sistemas de tipo *Workflow's* y sus características más importantes.

Un sistema *Workflow* es un sistema para gestionar y automatizar procesos de negocios. Se considera un tipo de sistema colaborativo ya que dentro de los procesos de negocio pueden participar múltiples usuarios y colaborar para cumplir sus trabajos. Aunque a veces no sea precisamente una colaboración en línea. Un sistema *Workflow* también se lo denomina BPM (en inglés, *Business Process Management*, Gestión de Procesos de Negocio).

'La automatización de un proceso de negocio, total o parcial, en la cual documentos, información o tareas son pasadas de un participante a otro a los efectos de su procesamiento, de

acuerdo a un conjunto de reglas establecidas.' -WIMC (*Workflow Management Coalition*).

Las características que suelen aportar estos tipos de sistemas son:

- Permitir colaboración en las tareas comunes de los participantes.
- Optimización de recursos humanos y técnicos.
- Automatización y eficiencia en los procesos de negocio.
- Asignación, seguimiento y notificaciones de las tareas del personal.

Gran parte de estos tipos de sistemas, no solo ayudan al manejo de los procesos de negocios, sino que también incorporan herramientas de colaboración que facilitan el trabajo de sus colaboradores como, por ejemplo, chats.

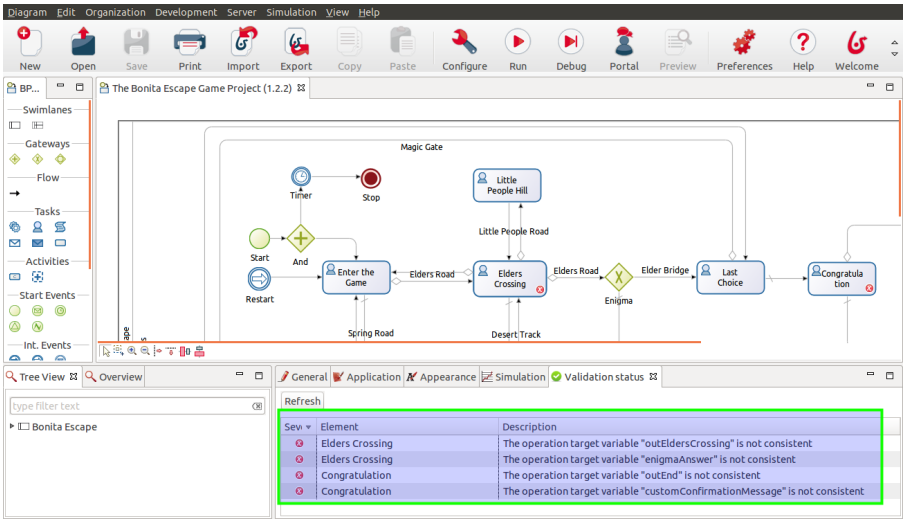

Figura 3.4: Bonita BPM, plataforma open-source de procesos de negocio.

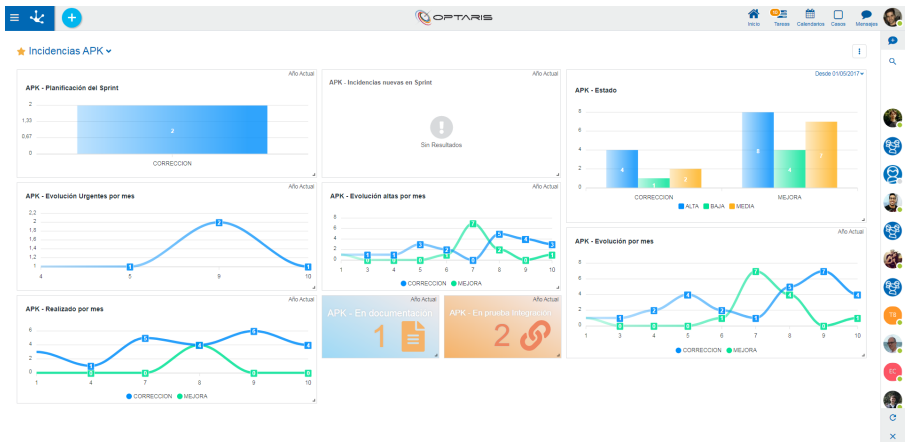

Figura 3.5: Deyel BPM, plataforma online, con herramientas colaborativas.

3.5. Actualidad de los Sistemas Colaborativos

Los sistemas colaborativos siempre han ido creciendo a medida que avanzan las tecnologías y las necesidades de las empresas. Pero en la actualidad y en una sorpresiva época de pandemia, la utilidad de los sistemas colaborativos ha aumentado exponencialmente, y su motivo es que muchas empresas u organizaciones han tenido que adaptarse a la situación de aislamiento, y seguir trabajando de forma remota. Por esta razón, los sistemas colaborativos han cubierto las exigencias que requiere este tipo de trabajo y se han convertido en sistemas necesarios en el día a día y en diferentes áreas.

Estos sistemas han demostrado estar a la altura de la situación y los beneficios que conceden. Un sistema colaborativo permite fortalecer los equipos de trabajos al permitir planificar, coordinar, comunicarse de manera eficiente, alinear fácilmente los objetivos y las prioridades para el éxito de los equipos desde cualquier lugar y de manera simultánea. En estos momentos, las empresas y las organizaciones deben fomentar la colaboración más que nunca. Con el fin de optimizar el trabajo diario, al igual que su productividad y sus objetivos.

En estos momentos, las empresas y las organizaciones deben fomentar la colaboración más que nunca. Con el fin de optimizar el trabajo diario, al igual que su productividad y sus objetivos. Una investigación de la consultora internacional *Gartner* [7] demostró que casi un 70 % de las empresas cree que el uso de las herramientas de colaboración aumenta entre un 20 % y un 40 % la productividad de sus trabajadores.

Capítulo 4

Metamodelo de Sistemas Colaborativos

4.1. Introducción

Previamente, hemos nombrado cual es el objetivo de un metamodelo y ya tenemos presente la utilidad de los sistemas colaborativos. En este capítulo, presentaremos el metamodelo de sistemas colaborativos que vamos a utilizar para crear un modelo, y luego aplicarle una transformación a través de la herramienta desarrollada en este trabajo.

El metamodelo de sistemas colaborativos empleado, fue creado por Luis Mariano Bibbó, profesor de la *Facultad de Informática de la Universidad Nacional de La Plata*, a través de su tesis *Modelado de Sistemas Colaborativos* [8] para su *Magíster de Ingeniería de Software*. Este metamodelo es una extensión de clases del conocido lenguaje de modelado UML [9], y nos permite modelar elementos genéricos, que son la base para poder crear cualquier modelo de un sistema colaborativo. En las próximas secciones, explicaremos la funcionalidad de cada uno de estos elementos.

Este capítulo, se enfocará en describir brevemente los elementos que componen este metamodelo para obtener una introducción de los mismos y facilitar la comprensión de los posteriores capítulos, en donde explicaremos la transformación de la herramienta. A lo largo de este capítulo, por cada sección usaremos ejemplos sobre un modelo de ajedrez, que también nos acompañará en los próximos.

4.2. Estructura de clases del metamodelo

En la figura 4.1, se puede observar toda la estructura de clases del metamodelo.

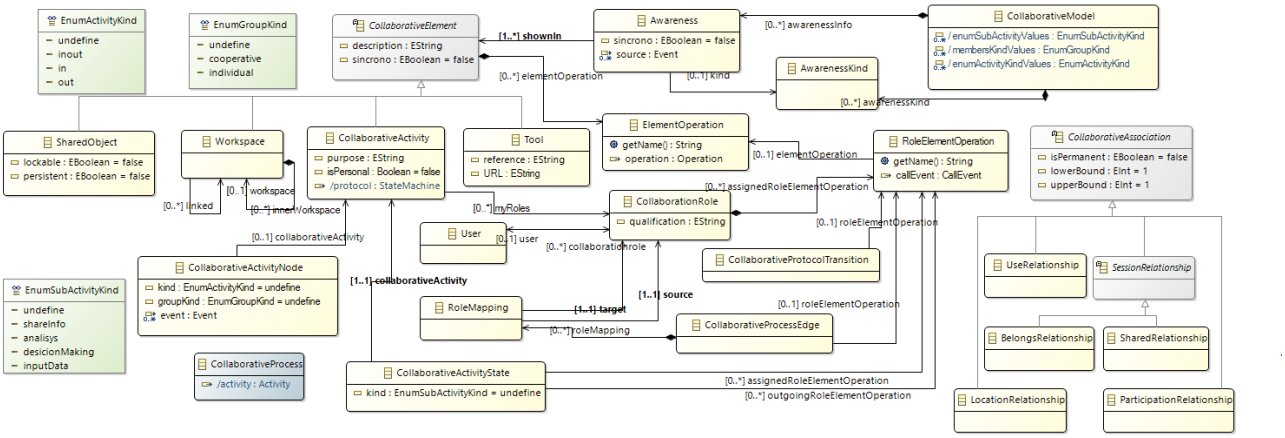

Figura 4.1: Estructura de clases del metamodelo colaborativo.

Estas clases nos permiten representar los objetos genéricos de un modelo colaborativo. Para crear estos objetos la herramienta de modelado provee diferentes tipos de editores gráficos, que facilitan la interacción con los modeladores para que puedan crear sus modelos de una forma ágil. En los siguientes capítulos, veremos cómo a partir de estas instancias de objetos se transforman, mediante nuestra herramienta de transformación, en un sistema web colaborativo.

No entraremos en detalle en todas las clases que muestra la figura, aquí solo nos concentramos en las clases que consideramos más relevantes para entender el funcionamiento de nuestra herramienta de transformación. Si se quiere obtener una información más detallada de cada clase, se recomienda leer la tesis del profesor Bibbó [8].

4.3. Elementos del metamodelo

En esta sección mencionamos los conceptos fundamentales de ciertos elementos del metamodelo. El objetivo de introducir estos conceptos es lograr tener una visión genérica de lo que representa cada uno de estos elementos. De esta forma, cuando comentemos el proceso de transformación de la herramienta, tenemos en claro el propósito del porque estos elementos son importantes en un modelo de un sistema colaborativo.

Los elementos colaborativos dentro del metamodelo están representados por la clase *CollaborativeElement*, aunque también existen excepciones de elementos que solamente se relacionan con esta clase, pero no heredan de ella. Por otro lado, estos elementos se asocian entre sí, a través de la clase *CollaborativeAssociation*, que se encarga de establecer las relaciones entre ellos. En los próximos apartados, veremos que existen diferentes tipos de asociaciones, y son una pieza clave en el modelado.

4.3.1. Operación

Una operación es el elemento más sencillo, y se caracteriza por simular la ejecución de una acción realizada por el usuario. Son representadas por la clase *ElementOperation* dentro del metamodelo y se asocian con los elementos colaborativos.

Los elementos colaborativos (clase *CollaborativeElement*) como lo son las actividades, las herramientas o los espacios de trabajo, son los elementos más relevantes del metamodelo, y como su nombre lo indica son los que permiten la colaboración de los usuarios, a través de sus interacciones. Pueden contener múltiples operaciones, por lo cual los usuarios al utilizar estos elementos pueden realizar las operaciones que cada uno proporciona. Más adelante, explicaremos cada uno de ellos.

En la figura 4.2, podemos ver, a través del editor de operaciones del metamodelo, las operaciones que se podrían modelar para el uso de un tablero de ajedrez.

Figura 4.2: Ejemplo de operaciones en tablero.

4.3.2. Espacio de Trabajo

Un espacio de trabajo o workspace es el entorno o área de trabajo donde ocurren las actividades colaborativas de los usuarios. Es exactamente igual a un espacio de trabajo real, por ejemplo, una oficina o un aula en donde los usuarios se encargan de realizar sus tareas. Los workspaces funcionan de una manera similar, con la excepción que en este caso son entornos virtuales.

Dentro del metamodelo colaborativo están representados por la clase *Workspace* y es considerado un elemento colaborativo, es decir que hereda de *CollaborativeElement*, heredando su estructura y comportamiento. De esta forma, como la clase *CollaborativeElement* puede tener una relación uno a muchos con *ElementOperation*, dentro de los workspaces podrán modelarse operaciones, que luego los usuarios podrán ejecutar. La figura 4.3, nos muestra un ejemplo de un espacio de trabajo con sus operaciones para el modelo de ajedrez.

Figura 4.3: Ejemplo de Workspace con operaciones.

4.3.3. Actividad Colaborativa

Una actividad colaborativa es la tarea o acción en la cual los usuarios colaboran para cumplir con sus objetivos. Al ser una acción, por lo general, su nombre está compuesto por un verbo, por ejemplo, en nuestro modelo de ajedrez, una actividad colaborativa podría ser "JugarPartida". Se pueden asociar con un espacio de trabajo, logrando que los usuarios que tengan acceso al mismo, vean la actividad en su interior. Dentro del metamodelo están representadas por la clase *CollaborativeActivity* y al igual que los *Workspaces*, son elementos colaborativos por lo cual también, su clase hereda de *CollaborativeElement*, permitiendo que se puedan definir operaciones en ellas.

Dentro del metamodelo, muchos elementos pueden asociarse con otros, y esto se realiza a través de lo que llamamos relaciones (*CollaborativeRelationship*). Cada relación tiene su obje-

tivo y la clase que la representa. En este caso, las relaciones entre las actividades y los espacios de trabajo, se las conoce como relaciones de pertenencia, y están identificadas con la clase *BelongsRelationship*. Cuando a través de los editores del metamodelo asociamos estos elementos colaborativos, internamente se crea una instancia de esta clase. En la figura 4.4, se puede observar la asociación entre un espacio de trabajo llamado "Mesa" y la actividad "JugarPartida".

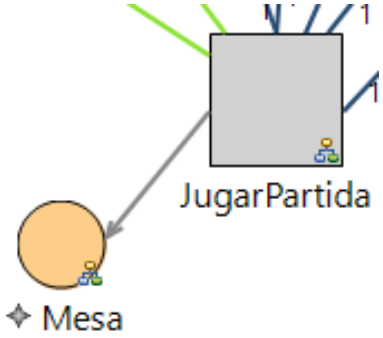

Figura 4.4: Vinculación entre workspace y actividad (*BelongsRelationship*) en el editor.

Las actividades son los elementos colaborativos más importantes del metamodelo, gran parte de la funcionalidad del sistema resultante gira alrededor de ellas. Albergan las herramientas que los usuarios usan para comunicarse y colaborar, y se encargan del manejo de la sincronización, tema que se verá en los próximos capítulos.

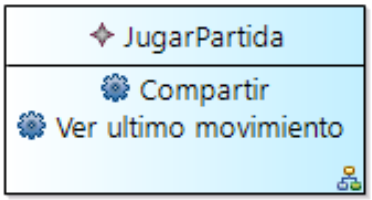

Figura 4.5: Actividad colaborativa con operaciones.

A su vez, las actividades colaborativas pueden contener múltiples estados, y en función de ellos se determina qué usuarios están colaborando y que operaciones de la actividad se pueden realizar en ese momento. Durante su ejecución y dependiendo de la interacción de los usuarios, la actividad podrá ir cambiando de estados, hasta que llegue una determinada acción que dé fin a su ejecución. Esta transición de estados se la conoce como protocolo de la actividad, y cada estado está representado dentro del metamodelo por la clase *CollaborativeActivityState*. En la figura 4.6, se puede observar un ejemplo de un protocolo de estados para nuestra actividad "JugarPartida" de nuestro ajedrez.

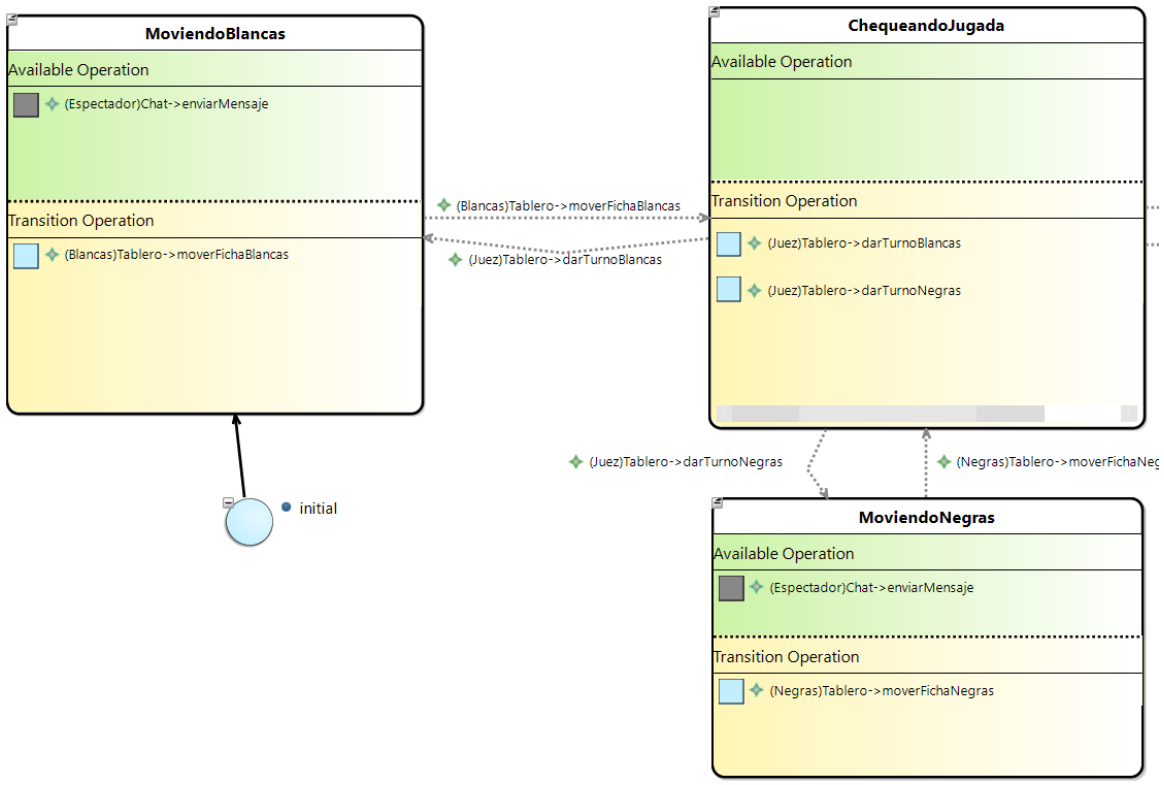

Figura 4.6: Ejemplo de protocolo de estados de la actividad "JugarPartida".

4.3.4. Herramienta

Las herramientas (*Tool*) dentro del metamodelo son los elementos que los usuarios usan para realizar sus actividades. Por ejemplo, en el ajedrez, para jugar una partida los usuarios usarán un tablero, o si se quieren comunicar lo pueden hacer mediante un chat.

Están representadas por la clase *Tool* y al igual que los elementos anteriores, esta misma hereda de la clase de los elementos colaborativos, *CollaborativeElement*, logrando obtener el comportamiento de las operaciones.

En toda actividad, por lo general hay al menos una herramienta, que permite al usuario cumplir con la actividad a través de ella. Por este motivo las herramientas se encuentran embebidas dentro de las actividades, y se asocian entre sí mediante una relación de uso, representada por la clase *UseRelationship*, que nos determina qué herramientas se usan dentro de la actividad.

En la figura 4.7, se presenta un ejemplo de dos herramientas, con sus operaciones, que pueden modelarse dentro de nuestro modelo de ajedrez.

Figura 4.7: Ejemplos de herramientas colaborativas con operaciones para un ajedrez.

4.3.5. Rol de Colaboración

Los roles de colaboración (*Collaboration Role*) comprenden un significado similar a los roles utilizados dentro de cualquier sistema informático. En un sistema, un rol define lo que el usuario puede realizar dentro del mismo. El rol se asocia con funcionalidades en las que el usuario tiene acceso, por lo cual determina que elementos el usuario puede visualizar y utilizar.

Dentro del metamodelo, un rol está representado por la clase *CollaborationRole*, y en conjunto con la clase *RoleElementOperation* (Operación del elemento del rol) se asocia con la clase *CollaborativeElement*, es decir, con los elementos colaborativos mencionados en las secciones anterior.

Los roles de colaboración nos ayudan a clasificar a los usuarios dependiendo de las responsabilidades o tareas que deben realizar, y nos permiten controlar el acceso a los elementos que cada tipo de usuario puede utilizar, según lo modelado. También, los roles nos establecen que usuarios participan en las actividades del modelo y esto se realiza a través de una nueva relación, llamada relación de participación (clase *ParticipationRelationship*). En la figura 4.8, se puede observar dentro del editor del metamodelo, múltiples roles asociados a nuestra actividad "JugarPartida" de nuestro modelo ejemplo.

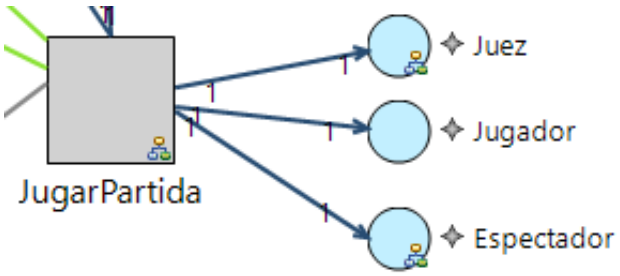

Figura 4.8: Ejemplo de relaciones de participación, en el editor, para la actividad "JugarPartida".

4.3.6. Awareness

El significado del término *Awareness* (conocimiento) es indicar que una persona se da cuenta o toma conciencia sobre algo. En nuestro caso, se encargan de mantener información sobre las actividades que realizan los usuarios en el sistema. Dentro del metamodelo, es un tipo de elemento que permite a los usuarios participantes conocer determinada información del resto de los usuarios. Por ejemplo, en nuestro modelo de ajedrez, los usuarios que juegan una partida, pueden ser conscientes que el otro jugador está conectado, o que realizó un determinado movimiento. Cualquier información del usuario que se quiera mostrar en línea, y a los demás participantes, forma parte de un *Awareness*. Son una pieza clave en los sistemas colaborativos, ya que ayudan a distinguir a los usuarios que la actividad se está realizando en simultáneo y de manera conjunta con otros usuarios.

En el metamodelo, están representados por la clase *Awareness*, y existe la posibilidad de crear tipos propios a través de la clase *AwarenessKind*. La clase *Awareness* mantiene un atributo llamado *showIn* que indica en qué elemento colaborativo se muestra determinado *Awareness*. Este atributo se puede asignar fácilmente cuando nos encontramos modelando mediante los editores del metamodelo, como lo muestra la figura 4.9.

4.3.7. Proceso Colaborativo

Cuando presentamos las actividades, mencionamos que tienen un protocolo de estados, que puede ir cambiando durante la ejecución de la actividad pero no nombramos como se inician ni cómo finalizan estas actividades. Esta funcionalidad es aportada por los procesos, ya que

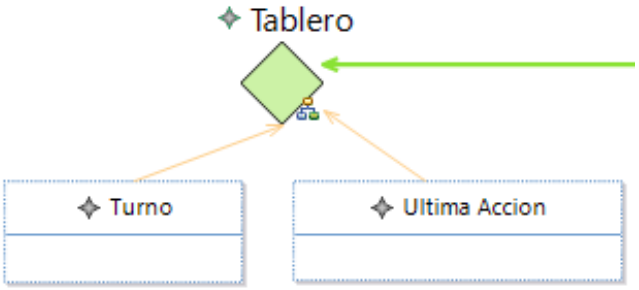

Figura 4.9: Ejemplo de *Awareness* en modelo de ajedrez.

se encargan de definir el orden de ejecución de las actividades, y además, son los puntos de entrada que tiene el usuario para iniciar el flujo de ejecución del sistema, según su participan en el mismo. Desde una visión más abstracta, los procesos son los encargados de definir los objetivos que tienen los usuarios al utilizar el sistema.

El metamodelo nos permite modelar múltiples procesos por cada modelo, y todas las ejecuciones de las actividades inician y finalizan según los procesos modelados. En el metamodelo, están representados por la clase *CollaborativeProcess* y tienen una relación directa con las actividades. En la figura 4.10, se puede ver un ejemplo de un proceso, para nuestro modelo de ajedrez, que estaremos explicando en detalle en los próximos capítulos.

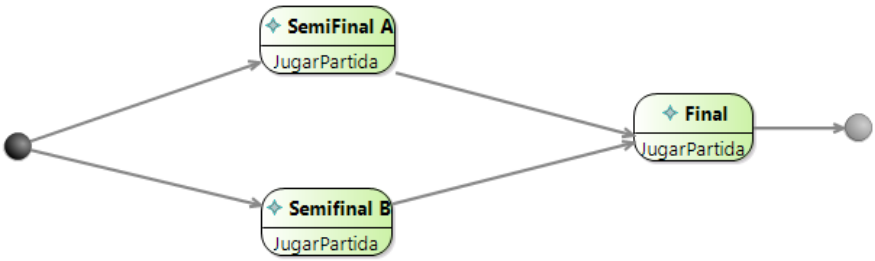

Figura 4.10: Ejemplo de proceso para un modelo de ajedrez.

Capítulo 5

Transformación Conceptual

5.1. Introducción

Ya conocemos los objetivos y funcionalidades de los elementos del metamodelo. En este capítulo vamos a conocer cuáles son sus representaciones. Estas representaciones las llamamos transformaciones conceptuales, y sus orígenes son parte de las decisiones que se fueron determinando en el transcurso de la etapa de análisis de este trabajo. El objetivo de estas decisiones es lograr obtener la mejor conceptualización posible, y que cada elemento del metamodelo sea representado de una forma correcta, tanto visualmente como funcionalmente.

La transformación consiste en convertir cada elemento del modelo en sus correspondientes representaciones, es decir, que se transforman en los objetos necesarios para que funcionen en un sistema destino, y cumplan con los propósitos para los que fueron creados. En este capítulo solo veremos la representación conceptual de cada uno, en los siguientes, se entrará en detalle en las representaciones técnicas.

Iremos de menor a mayor, partiendo de las representaciones más básicas a las más complejas. Durante este recorrido, presentaremos figuras de estas representaciones usando como ejemplo el modelo del juego de ajedrez visto previamente.

5.2. Transformación Conceptual de Operaciones

Su representación genérica es una "acción" que realiza el usuario mientras interactúa con el sistema. Como es un elemento que depende de la interacción, podría ser representado con un botón. Pueden existir operaciones que no precisamente se deben representar con un botón, por ejemplo, podría ser otro evento por parte del usuario como presionar una tecla. Pero estas decisiones dependen del modelador que utiliza la herramienta. En estos casos, se podría modificar el código del sistema resultante una vez aplicada la transformación. Para una mejor facilidad y con fines demostrativos, todas las operaciones que existen en el modelo, son convertidas en botones, una vez que la herramienta aplicó la transformación.

Más adelante, se explicarán los distintos tipos de operaciones que pueden existir dentro del resultado final de la herramienta. Ya que existen operaciones en donde sus comportamientos dependen del modelador, como también operaciones que determinan las transiciones de las actividades dentro de los procesos. El comportamiento de estas últimas, está dado por lo modelado y por la misma transformación.

Figura 5.1: Representación gráfica de operaciones dentro de una herramienta nombrada "tablero".

En la figura anterior, podemos observar dos operaciones, "darTurnoBlancas" y "darTurnoNegras", ambas modeladas en nuestro modelo de ajedrez. A simple vista son solo botones, pero cuando entremos más en detalle veremos cómo se representan dentro de la tecnología seleccionada, y la importancia que tienen dentro del sistema resultante. Recordemos que este capítulo solo veremos la representación gráfica y funcional, el comportamiento será visto más adelante.

5.3. Transformación Conceptual de Espacios de Trabajo

Como mencionamos en el capítulo anterior, un workspace es el entorno donde ocurren las actividades colaborativas de los usuarios. Por este motivo, su transformación conceptual puede verse como un contenedor que contiene operaciones y actividades.

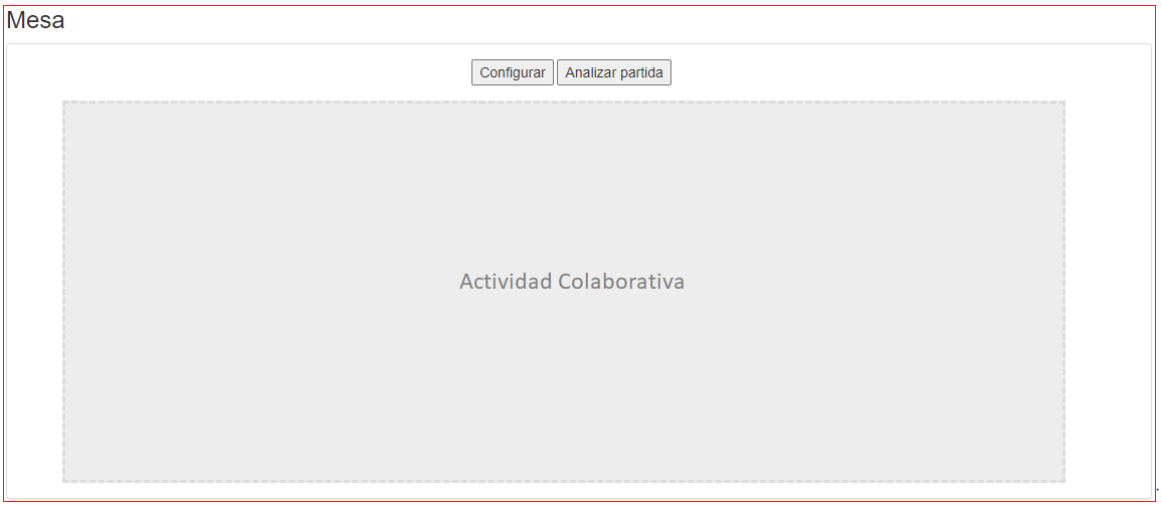

Figura 5.2: Ejemplo de una transformación conceptual de un workspace.

En la figura 5.2, se puede observar, dentro del marco rojo, una transformación conceptual de un workspace que simula ser una "Mesa" en nuestro modelo y será el entorno de trabajo, y en donde se ejecutarán las actividades colaborativas que se modelan dentro del mismo. El metamodelo permite crear los workspaces que se deseen y asociarle múltiples operaciones. En el ejemplo, se muestran dos operaciones dentro, "Configurar" y "Analizar partida", operaciones con las que el usuario interacciona con el sistema. Más adelante veremos el funcionamiento de los workspaces y los componentes de las tecnologías que se generan en el sistema resultante.

5.4. Transformación Conceptual de Actividades Colaborativas

Como las actividades suelen ocurrir en entornos de trabajo, los workspaces son los encargados de contenerlas. Al contener operaciones la transformación conceptual de una actividad colaborativa, será un nuevo contenedor que tendrá en su interior sus operaciones y herramientas, como lo muestra la siguiente figura.

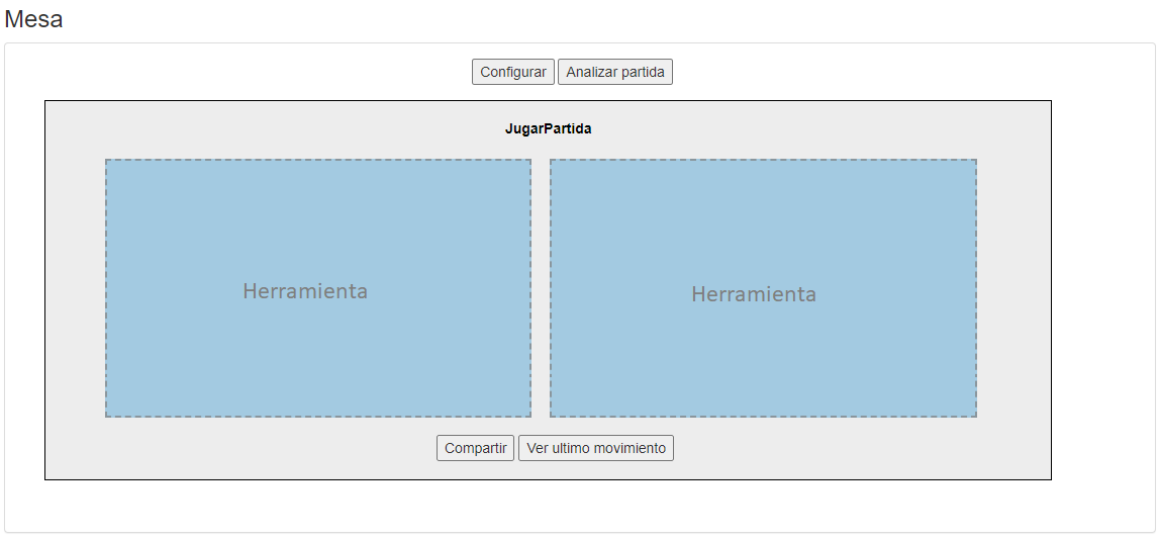

Figura 5.3: Ejemplo de la transformación conceptual de una actividad colaborativa, nombrada "JugarPartida".

Hemos mencionado también que las actividades pueden contener estados, y que estos estados pueden ir cambiando a medida que los usuarios realicen sus operaciones. Las actividades siempre van a estar acompañadas de un estado y en ocasiones puede ser necesario que el modelador, o el mismo usuario que usa el sistema, deba ver el estado en el que se encuentra. Por consiguiente, la transformación conceptual de un estado de una actividad colaborativa podría ser una etiqueta que acompañe el nombre de la actividad, para determinar el estado actual.

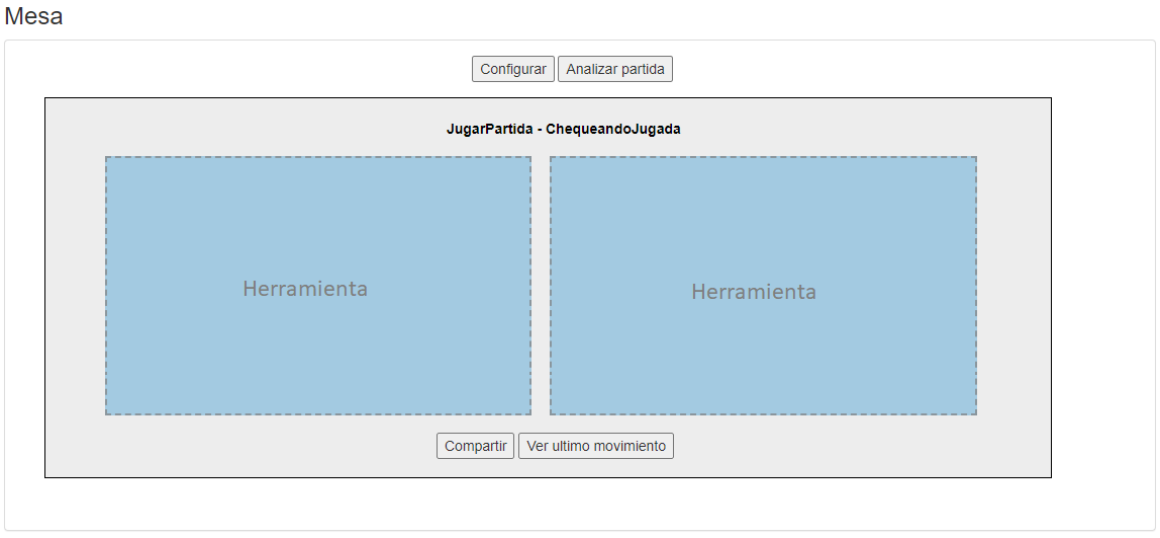

Figura 5.4: Ejemplo de transformación conceptual de una actividad con estado.

En el ejemplo de la figura 5.4, se puede observar el estado "ChequeandoJugada" para la actividad "JugarPartida" de nuestro modelo de ajedrez.

5.5. Transformación Conceptual de Herramientas

Como todo elemento colaborativo, la transformación conceptual de las herramientas son contenedores, ya que definen operaciones. Recordemos que los objetivos de los contenedores es indicarle al modelador donde se encuentra el elemento modelado, indicando el nombre, las operaciones asociadas, y una distinción a través de un color. Al mismo tiempo, le permitirá reconocer el componente creado dentro de la tecnología, tema que abordaremos en los próximos capítulos. De esta forma, el modelador, que use la herramienta de transformación, podrá distinguir en qué archivos o componentes puede incluir o expandir la lógica, para adecuar el sistema a su gusto.

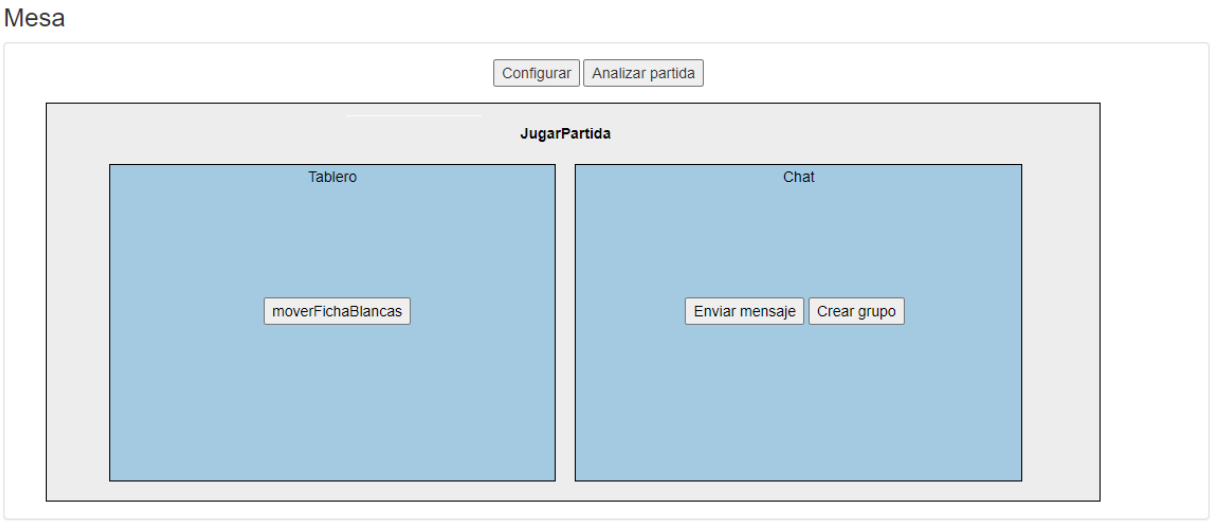

Figura 5.5: Ejemplo de transformación conceptual de herramientas.

En la figura 5.5, se pueden observar, en recuadros de color celeste, dos ejemplos de herramientas para nuestro modelo. Las dos se encuentran embebidas dentro de la actividad "JugarPartida", la primera representa un tablero de ajedrez y la segunda una herramienta de chat, ambas con sus respectivas operaciones.

5.6. Transformación Conceptual de Roles de Colaboración

El rol es un elemento que determina las actividades y responsabilidades que están relacionadas a los usuarios. Los roles sirven a las tareas para especificar quién las realiza. En nuestro caso, un rol define los elementos que el usuario verá y que podrá emplear cuando inicie sesión en el sistema resultante. No nos olvidemos que en este capítulo estamos exponiendo los conceptos genéricos y representativos de los elementos utilizados en la transformación. Por ende, considerando e interpretando el significado del rol, se decidió vincularlo al usuario participante.

Es un concepto fundamental ya que sin él, cualquier usuario podría acceder a todos los elementos colaborativos del sistema. En los próximos capítulos se desarrollará más ampliamente este tema, pero ahora basta con saber que el rol se convierte en un atributo más que acompaña al usuario. Este atributo es único por participante, pero veremos que puede llegar a cambiar dependiendo del tipo de rol y la colaboración del usuario dentro de los elementos colaborativos.

En el sistema resultante, siempre que el usuario inicie sesión se mostrará el rol al que pertenece como se muestra en la siguiente imagen.

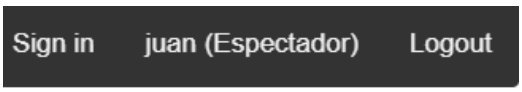

Figura 5.6: Ejemplo de visualización del rol "Espectador"del usuario.

5.7. Transformación Conceptual de Awareness

Los awareness son los elementos encargados de mostrar información de lo que realizan los usuarios, y esto es una característica muy importante dentro de un sistema colaborativo. Más adelante se comentará con qué tecnología se abarcan estos componentes y se entrará más en profundidad sobre el tema. En esta sección, como lo venimos haciendo en todo el capítulo, vamos a mencionar cuál es el concepto de un awareness, es decir cómo se realizará la representación de estos elementos una vez aplicada la transformación.

Como es un elemento bastante más complejo que los anteriores, y nuestro interés es explicar

su transformación de forma genérica y global, vamos a describir nuestra idea con un simple ejemplo. Supongamos que somos un modelador, y estamos usando nuestra herramienta para crear un sistema colaborativo a partir del metamodelo. Una vez que terminamos de modelar los elementos básicos queremos agregar un awareness, al que llamaremos “Presencia” y se muestre en una determinada herramienta (por ejemplo un “Chat”) y nos permita visualizar los usuarios que están actualmente en línea usando esa herramienta. Ahora bien, queremos despreocuparnos de la funcionalidad de nuestro awareness, que sea autónomo es decir que sepa sincronizarse cuando un usuario está en línea y cuando no, y además que se realice en tiempo real.

Ahora analicemos una posible implementación, en la que se podría llegar a convertir nuestro awareness “Presencia”. Sabemos que va a ser un listado de usuarios, que se sincronizará cuando algún usuario se conecte o se desconecte. Más allá de cómo se visualicen, tanto los usuarios conectados como los desconectados, podemos pensar que el listado debe cambiar cuando un usuario entra o sale de la herramienta “Chat”, es decir que se agregará o se quitará del listado de usuarios. Por lo tanto, podemos interpretar que estas acciones implícitas que el usuario realiza son eventos que automáticamente se actualizarán en base a lo que hacen los demás usuarios.

Más adelante, veremos que existen distintos tipos de awareness que también generarán otros eventos. Pero todos los awareness se convertirán en eventos automáticos. También veremos que la ejecución en tiempo real de estos eventos será dada por la tecnología que fue seleccionada para soportar esta característica, así que por el momento no es necesario explicarla en este capítulo. Lo que se visualizará depende tanto del modelo como de las modificaciones posteriores que realice el modelador, pero aquí basta con saber que el elemento se sincronizará automáticamente cuando surja algún cambio. Por lo tanto, un awareness es un contenedor que se encarga de sincronizar los eventos de los usuarios que pueden verlo.

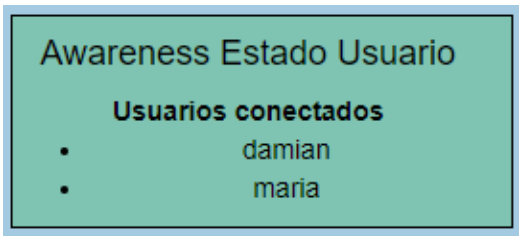

Figura 5.7: Ejemplo de transformación conceptual de awareness ”Presencia”.

5.8. Transformación Conceptual de Procesos Colaborativos

Los procesos son los elementos más complejos, y se encargan de darle vida a la interacción de los usuarios, indicando el orden de ejecución de las actividades y siendo los puntos de entrada a dicha ejecución. Por esto, la representación gráfica de un proceso es un poco distinta al resto de los elementos. Al ser los puntos de partida a la ejecución podemos decir que una transformación conceptual de un proceso podría ser un enlace, que gráficamente podría visualizarse como una opción dentro de un menú o submenú, como lo muestra la siguiente figura.

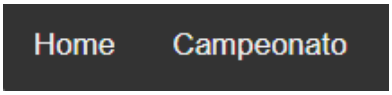

Figura 5.8: Ejemplo de transformación conceptual de un proceso “Campeonato” como punto de entrada.

Si el usuario interacciona con este menú, podría dar inicio a la ejecución de un “Campeonato”, iniciando la primera actividad del proceso. Por otro lado, el orden de ejecución de las actividades, que está dada por los procesos, no es algo que sea representado gráficamente, sino que es una parte del comportamiento interno del sistema que se desea modelar. En los próximos capítulos, cuando se presente la transformación real explicaremos el funcionamiento del comportamiento del sistema resultante.

Capítulo 6

Tecnologías

6.1. Introducción

Este trabajo está enfocado en la automatización y en la colaboración de los usuarios en línea. Teniendo en cuenta estos dos conceptos, surgió la necesidad de buscar tecnologías que permitan proporcionar la sincronización y la comunicación entre los componentes del sistema y facilitar la generación de código automático. Por este motivo, en este capítulo, se presentan las tecnologías que fueron elegidas y el motivo por el cual fueron seleccionadas, para aprovechar los beneficios que cada una de ellas otorga. Así también, se especifica en qué parte del proceso, de la creación de la herramienta, se utilizan.

Primero, se describe la arquitectura integral que tendrá la herramienta final una vez que se haya aplicado la transformación. En este caso, la arquitectura elegida es cliente-servidor, en donde se comentará los privilegios que aporta y el motivo de su elección. A su vez, comentaremos API Rest, tecnología que se encargará de aportar una comunicación entre el cliente y el servidor.

Luego, se exhibe TypeScript que fue el lenguaje de programación principal elegido. Es la base para las tecnologías que se irán presentando a lo largo del capítulo, ya sea para el cliente como para el servidor.

Una vez presentado el lenguaje global de la herramienta, se exponen los frameworks primordiales. Angular, como framework del cliente y Express como framework del servidor. En ambos frameworks se añadieron complementos que también se detallan en cada sección.

Más tarde, se detalla MongoDB como base de datos no relacional, y el porqué fue elegida sobre los otros tipos de base de datos.

Como se mencionó al principio de esta sección, el sistema final es una herramienta colaborativa con usuarios en línea, por lo cual, se debe lograr una sincronización entre los clientes y el servidor. Para abarcar esta problemática, se decidió emplear el uso de WebSockets. Una de las tecnologías más relevantes de este desarrollo y la que estará presente en el resto de los capítulos. Por último, se presenta Acceleo, herramienta que nos facilita la transformación, ya que se encarga de interpretar nuestro modelo y crear los archivos necesarios, que contienen el código que le da funcionalidad al sistema resultante.

6.2. Arquitectura Cliente-Servidor

La herramienta está pensada para que futuros programadores tengan a partir de un modelo, creado con el metamodelo colaborativo, un sistema web parcial de sus requerimientos. Sabiendo que apuntamos a usuarios que son desarrolladores, nuestro sistema web debe tener una arquitectura reconocida y que a su vez aporte una estructura funcional. Por este motivo, se seleccionó cliente-servidor como arquitectura principal del sistema web. Además, es un modelo de diseño bastante maduro y que aporta ventajas ya distinguidas, que se mencionan en los próximos apartados. No se realizará una explicación muy detallada sobre este tema, ya que escapa a lo relacionado con el desarrollo. Pero si es importante conocer el concepto y tenerlo en cuenta, ya que se mencionan sus componentes de forma reiterativa en los próximos capítulos. Si se quiere obtener una descripción más detallada del tema, en las referencias bibliográficas se deja un enlace relacionado a esta arquitectura.

La arquitectura Cliente-Servidor [10] es un modelo de diseño de software en donde se dividen las tareas en dos componentes. El componente servidor, que es el que proporciona la información, y el componente cliente, que es el consumidor de la información, también llamado consumidor de servicios. Por lo general, existe un solo servidor, aunque pueden existir varios para múltiples clientes. Ambas partes interactúan entre sí a través de internet para obtener y procesar los datos que el usuario requiere. La figura 6.1 muestra un ejemplo de la arquitectura descrita.

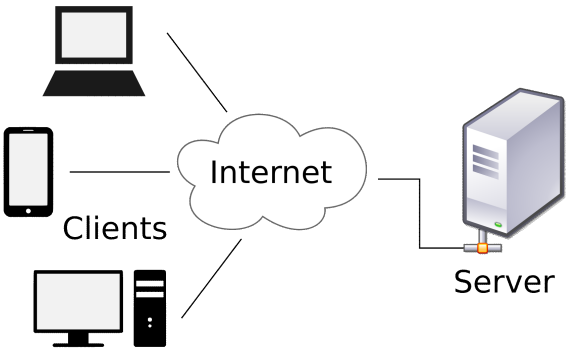

Figura 6.1: Arquitectura Cliente-Servidor.

Este diseño de software se puede considerar un modelo distribuido, ya que se divide el procesamiento entre las diferentes partes. Por lo general, los clientes son computadoras personales o cualquier dispositivo con acceso a internet. En cambio, los servidores son computadoras más específicas y con mejores prestaciones, ya que deben soportar múltiples solicitudes por parte de los clientes.

Para proveer el o los servicios, el servidor ejecuta programas que están continuamente en ejecución a la espera de solicitudes de los clientes y es quien mantiene toda la lógica del negocio y la comunicación con la base de datos, que habitualmente reside en el mismo servidor.

Como se expuso en la introducción del capítulo, esta arquitectura fue elegida para este desarrollo ya que es una de las predominantes actualmente, muy conocida por los programadores, y además, porque mantiene una serie de ventajas que hacen que sea la más utilizada.

A partir de ahora y en adelante, nos referiremos al lado del cliente o frontend de manera indistinta, como así también servidor o backend.

6.2.1. Ventajas

- **Centralización de los recursos y la integridad de los datos:** Esta tarea está bajo la responsabilidad del servidor, que es el encargado de mantener la consistencia de los datos y el uso adecuado de los recursos. De esta forma, si un cliente falla o es defectuoso no afectará al sistema general.
- **Fácil mantenimiento:** Ambas partes se manejan por separado, lo que facilita el mantenimiento específico de cada una de ellas, sin que afecte a la otra.

- **Escalabilidad:** Se puede aumentar el número de clientes o servidores sin ningún problema, y adaptarse sin perder calidad.
- **Tecnologías avanzadas:** Es una arquitectura bastante madura, y a lo largo de los años fueron surgiendo tecnologías en base a este esquema, que permiten la seguridad de los datos, la interacción con el usuario y la eficiencia de los recursos.

6.3. API Rest

Ya conocemos las partes de la arquitectura, ahora exponemos cómo se van a comunicar entre sí. El intercambio de datos entre el cliente y el servidor se decidió realizar a través de API Rest.

API Rest se define como una interfaz, entre sistemas, que utiliza el protocolo HTTP [11] para obtener los datos o advertir la ejecución de alguna operación, usando un formato XML [12] o JSON [13]. API Rest es una alternativa a protocolos de intercambio de datos más complejos como SOAP [14].

En el presente no hay aplicación web que no utilice Rest, para el intercambio de datos. Y desde su surgimiento, sus usos se fueron ampliando, de tal forma que con el tiempo se convirtió en un estándar, siendo la tecnología para la creación de APIs más lógica, eficiente y habitual para servicios de Internet.

Las peticiones HTTP contienen toda la información necesaria para procesarlas. No es necesario que el cliente ni el servidor mantengan un estado. El cliente le dice al servidor que operación quiere ejecutar dependiendo del tipo de solicitud HTTP enviada. Por ejemplo, GET corresponde a leer y consultar un dato, POST a crear un dato, PUT o PATCH a modificar un dato, y DELETE a eliminar. Los objetos se manipulan a partir de una ruta o URI. Una URI es un identificador único que representa un recurso dentro de la API. El cliente indica al servidor, a través de la URI, a que recurso quiere aplicarle las operaciones previamente nombradas.

6.3.1. Ventajas

- **Separación entre el cliente y servidor:** Separa completamente la interfaz del usuario con el procesamiento y almacenamiento de datos. Mejorando la portabilidad, escalabilidad

y evolución independiente de las partes.

- **Visibilidad, fiabilidad y escalabilidad:** Cualquier equipo de desarrollo futuro que quiera trabajar sobre el sistema podrá escalar y extender el software sin problemas. Mantener separado el frontend y el backend hace que el sistema web sea más flexible.

- **Independiente de la plataforma:** Siempre se adapta a la sintaxis y plataforma donde se esté trabajando. Dando mayor libertad a la hora de cambiar y probar nuevos entornos. La única restricción es que se respete el formato de los datos que se envían, XML o JSON.

- **Principio de HATEOAS [15]:** Cada API Rest debe cumplir este principio, que nos dice que en cada respuesta del servidor, parte de la información que contendrán deberán ser hipervínculos de navegación a otros recursos del cliente.

- **Uso de Hipermedios:** Concepto de los sistemas webs que permite que el usuario pueda navegar por un conjunto de recursos a través de enlaces HTML [16].

6.4. TypeScript

Como nuestro trabajo se basa en la metodología de desarrollo dirigida por modelos, la herramienta se encargará de crear código automáticamente. Por esta razón, necesitamos que el código esté escrito en un lenguaje de programación sencillo, liviano, que no requiera de una configuración complicada y que no sea compleja su ejecución. Además, cómo nuestros usuarios finales son programadores, tenemos que buscar un lenguaje medianamente conocido. Teniendo en cuenta las pautas nombradas anteriormente, nuestro foco se orientó hacia JavaScript [17]. Con el fuerte crecimiento que viene demostrando en estos últimos años, por el surgimiento de tecnologías como NodeJs (explicado en las siguientes secciones) y los nuevos frameworks del lado del cliente como Angular o React, hacen que JavaScript sea una interesante propuesta. Asimismo, es un lenguaje de programación dinámico, simple, que no requiere una puesta a punto tediosa, su ejecución es casi inmediata y la gran mayoría de los desarrolladores lo conoce. Es decir, que cumple con todos los requisitos que necesitamos para generar código automáticamente. Y si a este lenguaje, le podemos agregar ventajas de un lenguaje tipado, sería lo ideal, ya que nuestro código producido aumentaría su calidad. Mejor aún, si logramos que este mismo

lenguaje se ejecute tanto del lado del cliente como del servidor, tendríamos un lenguaje global entre las aplicaciones. Y ahí, es donde surge TypeScript, que veremos a continuación.

TypeScript [18] es un lenguaje de programación basado en JavaScript, es decir que extiende su sintaxis. Dicho con otras palabras, es un JavaScript evolucionado, ya que no solo aporta los beneficios de un lenguaje dinámico, sino que añade tipado estático y permite crear estructuras de clases, entre otras ventajas que se nombraran posteriormente. Estas son características que no se presentan en el lenguaje JavaScript original.

TypeScript, es libre y de código abierto creado y mantenido por la empresa Microsoft. Otro de los motivos por el cual es elegido para este desarrollo, ya que no requiere de licencias pagas. Previamente se nombró que TypeScript puede ejecutarse en las dos partes de la arquitectura, en el cliente y en el servidor. Este beneficio surgió por la misma evolución de JavaScript, por las nuevas tecnologías y fundamentalmente por los nuevos frameworks. Esta ventaja se aprovechó al máximo en este trabajo.

El desarrollador va a obtener un sistema web realizado completamente con TypeScript, y sin ningún inconveniente, va a poder comenzar a implementar su código.

6.4.1. Ventajas

- **Compilado:** Al ser un lenguaje compilado/transpilado, ayuda al programador final a detectar errores antes de la ejecución.

- **Programación Orientada a Objetos:** Si bien en JavaScript existe algo parecido, llamados Prototype [19], carece de muchas características de este tipo de programación. Hay que ingeniarse algunas soluciones, para que se comporte de la misma manera. En TypeScript, esto no sucede, ya que recopila lo suficiente para ser un lenguaje orientado a objetos, como por ejemplo, nos provee de clases, interfaces, herencia y composición, entre otras características.

- **Moderno:** Se trata de un lenguaje de programación actual, con opciones interesantes, como permitir que una variable pueda tener diferentes tipos, decoradores, anotaciones y soporte de tipos genéricos.

- **Aparición de Frameworks:** Muchos fabricantes apuestan en la creación de frameworks [20] basados en el lenguaje TypeScript, como Angular o Ionic. Garantizando el futuro del lenguaje.

- **Servidor:** El surgimiento de NodeJs, plataforma basada en JavaScript que se verá más adelante, nos permite crear aplicaciones del lado del servidor que soportan este lenguaje, abriendo otros caminos a nuevas opciones.

- **Flexibilidad:** JavaScript es un lenguaje muy flexible, es decir no es necesario la definición de tipos a las variables. TypeScript, en cambio, exige la declaración de tipos a cada variable. Pero en algunos casos, es necesario mantener la flexibilidad de JavaScript, y esto se puede realizar por ejemplo usando el tipo genérico any. De esta forma, TypeScript se vuelve igual de flexible que JavaScript, y es una gran ventaja, poder mantener las características de ambos lenguajes.

6.5. NodeJs

Para comentar y explicar el framework del backend Express, y el framework del frontend, Angular, antes debemos pasar brevemente por la definición de NodeJs.

NodeJs o Node [21] es un entorno de ejecución multiplataforma y de código libre, basado en JavaScript, construido con el motor JavaScript V8 de Google (usado también en Google Chrome) y que comúnmente se ejecuta del lado del servidor. Su objetivo es proporcionar facilidades para crear aplicaciones altamente escalables. Además, su arquitectura está focalizada en el manejo concurrente de las conexiones, en lugar de crear nuevos hilos para cada conexión como lo haría cualquier servidor. En este caso, cada conexión dispara eventos dentro del motor de Node. De esta forma, el servidor que lo soporte, garantiza decenas de miles de conexiones concurrentes y siempre estará ejecutándose, porque no genera bloqueos. Nuestra herramienta generará sistemas colaborativos con una cierta cantidad de usuarios en línea, entonces tenemos que garantizar que las conexiones sean ligeras y es la razón por la cual se eligió este tipo de entorno, con el adicional de la sincronización de los eventos a través del lenguaje JavaScript.

Un dato no menor, es que los conceptos de JavaScript son los mismos, para el desarrollo, en el lado del cliente como en el servidor. NodeJs ofrece un conjunto de APIs adicionales

para soportar funcionalidades que son útiles sin navegador web, como por ejemplo para crear servidores y servicios que siguen el protocolo HTTP o para acceder al sistema de archivos.

6.5.1. Ventajas

- **Excelente rendimiento:** Node fue diseñado para mejorar el rendimiento y la escalabilidad de las aplicaciones web. Es de gran ayuda para sistemas de tiempo real.
- **Código en JavaScript:** es JavaScript puro, lo que significa que es mucho más rápido que otros lenguajes. Además, no requiere intercambio de lenguaje, ya que se ejecuta código JavaScript en el cliente y el servidor.
- **TypeScript:** Existe la posibilidad de usar TypeScript, con todos los beneficios que se comentaron previamente.
- **NPM [22] (del inglés, Node Package Manager):** El manejador de paquetes de Node permite instalar miles de paquetes reutilizables. Presenta también resolución de dependencias y automatización de herramientas de compilación.
- **Portable:** Funciona para versiones de los siguientes sistemas operativos, Windows, OS X, Linux, Solaris, FreeBSD, OpenBSD, WebOS y NonStop OS.
- **Comunidad:** Los desarrolladores de terceros son muy activos, y la comunidad siempre está dispuesta a ayudar.
- **Frameworks:** Con el tiempo surgieron frameworks, para clientes y servidores, basados en este entorno. Por lo cual, nos permite aprovechar todos los beneficios que aporta esta plataforma en ambas partes de nuestra arquitectura.

6.6. Angular

Angular es el framework elegido para el lado del cliente. Es un framework basado en TypeScript, creado por Google, que nos permite crear aplicaciones web SPA [23] (de sus siglas en inglés, Single-Page Application, Aplicación de una sola página), este término se refiere a aplicaciones que solo se cargan una vez. Es decir, que mientras el usuario interactúa con el

sistema, no se tiene que recargar todos los componentes, solo se cargan los necesarios, de esta forma se obtiene una mayor fluidez y una mejor navegación.

Por otro lado, necesitamos mantener una estructura adecuada para los archivos, que contendrán el código que se genere en el frontend, y que esté familiarizada con la mayoría de los desarrolladores. Este es otro punto a favor de Angular, y uno de los motivos de porqué fue seleccionado para este trabajo, ya que nos aporta la arquitectura MVC [24] (Model-View-Controller, Modelo-Vista-Controlador) para estructurar nuestra herramienta y facilitarnos el trabajo. Angular está basado en componentes, que son pequeñas porciones de la página (plantillas) y que están asociadas a un controlador. Estos componentes pueden manejarse individualmente y reutilizarse dentro de otros. De esta forma, obtenemos un mayor nivel de abstracción y modularización.

Además, es un framework moderno, conocido, de fácil configuración y que nos posibilita la creación de aplicaciones web escalables, ya que nos otorga un conjunto de herramientas para desarrollo muy completo.

6.6.1. Ventajas

- **TypeScript como lenguaje de programación:** En Angular se pueden usar diferentes lenguajes de programación, pero prioriza TypeScript, aprovechando al máximo sus ventajas.
- **Creación de sistemas web dinámicos:** Mejora la interacción y la comunicación con el usuario gracias a la arquitectura SPA.
- **Facilidad de mantenimiento:** Sus módulos, los componentes, las vistas y los servicios están acoplados de tal forma, que facilitan el mantenimiento futuro del software.
- **Modularización y abstracción:** Si hay algo que no falta en Angular es la modularización y la abstracción, ya que el objetivo del framework es mantener siempre el código lo más reducido posible. Este punto, se relaciona directamente con el anterior, porque generar una buena modularización ayuda al mantenimiento.
- **Optimización y eficiencia:** Está altamente optimizado para la carga de las plantillas y gracias a su enrutador, solo se carga el código necesario favoreciendo la eficiencia global

del sistema.

- **Mejora la productividad:** Los elementos de Angular están preparados para su creación y manipulación de forma inmediata. Se pueden crear vistas de interfaz, componentes, servicios, entre otros, de una manera eficaz. Mediante la herramienta de línea de comandos (Angular CLI) se permite crear, eliminar o modificar los elementos previamente nombrados, e incluso crear test unitarios. Gracias a esta herramienta el comienzo del desarrollo en el framework se hace muy provechoso.

6.7. Express

En la tercera sección de este capítulo se comentó que el frontend y el backend se van a comunicar a través de una API Rest. Del lado del frontend, Angular posee servicios para hacer llamados HTTP al servidor y cumplir con su parte. En el backend, necesitamos un framework que nos permita crear nuestra API Rest, como punto de entrada a los datos. Para este trabajo, se optó por Express.

Express es un framework de código abierto, minimalista y el más popular de Node, ya que se ejecuta sobre él. Está construido para crear aplicaciones webs y APIs. En nuestro caso, sólo lo usaremos para crear una API, que luego el cliente consumirá.

Cuando hablamos de un framework minimalista, nos referimos a que contiene lo básico y necesario para cumplir su objetivo. De hecho, es un software muy optimizado, pero a simple vista, al ser minimalista, podríamos pensar que carece de muchas características. Pero cómo Express es un software de código abierto, los desarrolladores, incluso los propios dueños del producto, fueron creando middlewares [25] compatibles para abordar cualquier problema típico de los sistemas webs. Por ejemplo, librerías para el manejo de sesiones, cookies, seguridad, entre otros. Haciendo que la creación de APIs con el framework sea más rápida y ágil.

En el próximo capítulo, mencionaremos las URIs [26] de la API que se crean cuando se genera el sistema resultante otorgado por la herramienta.

6.7.1. Ventajas

- **Manejo de peticiones HTTP:** Fácil manejo y creación de rutas que soportan los diferentes métodos del estándar del protocolo HTTP (POST, DELETE, PATCH, GET, entre otros).
- **Middleware:** Se pueden agregar librerías en cualquier punto del software. Para la autenticación de los usuarios, se aprovecharon el uso de middlewares y herramientas novedosas y populares para facilitar el desarrollo, como Passport y JWT.
- **Flexibilidad:** Al estar ejecutando sobre Node, el framework absorbe los mismos beneficios que Node obtiene de JavaScript.
- **Optimización:** Los módulos del framework son reducidos y están completamente optimizados.
- **Integración:** Se puede integrar con otros motores de renderización de vistas.
- **Configuración:** Se pueden configurar, de una manera muy simple, los módulos pertenecientes.

6.8. MongoDB

Como base de datos se usará MongoDB, que es una base de datos no relacional construida con C++ y orientada a documentos. Es decir, que la información no se mantiene en tablas o registros, sino que se guardan en documentos. Estos documentos pueden ser en formato JSON [13], que se almacenan en su formato binario llamado BSON [27].

La gran diferencia de esta base de datos con respecto a las de tipo relacional, es que no se sigue un esquema. Los documentos de una misma colección, concepto similar a las tablas de las bases de datos relacionales, pueden tener diferente formato.

Se optó por este tipo de base de datos por tres motivos, primero nuestra API Rest manejara colecciones en formato JSON, de esta forma evitamos que se realicen conversiones de tipos, mejorando el rendimiento del sistema web. Segundo, al ser una base de datos para almacenar información semi estructurada, es ideal para las estructuras que se generan cuando se aplica

la transformación del modelo. Y por último, es de fácil integración, configuración y contiene drivers para el lenguaje de programación JavaScript.

6.8.1. Ventajas

- **Aplicaciones escalables:** Permite a través de sus opciones de replicación y fragmentación, que las aplicaciones puedan escalar horizontalmente sin demasiados inconvenientes.
- **Rendimiento:** Los datos se almacenan rápidamente, ya que soporta las mismas colecciones del lenguaje de programación. y las consultas están optimizadas para el formato JSON.
- **Consola:** MongoDB provee una consola que nos permite ejecutar comandos (o consultas) y también se pueden usar muchas funciones propias de JavaScript.
- **Soporte:** Existen drivers para soportar los más conocidos lenguajes de programación, como C#, Java, JavaScript, PHP, Python, NodeJs, C, C++, Perl, Ruby o Scala.
- **Funciones de base de datos:** Proporciona consultar ad-hoc, indexación, replicación, fragmentación, balanceo de carga, agregación, entre otras.

6.9. WebSocket

Bien, ya hemos presentado las tecnologías del lado del cliente, del servidor y detallamos cómo se comunicaran entre sí. Nuestra API Rest nos ayudará a consumir datos de la base de datos desde el lado del cliente, a través de solicitudes HTTP. Con las rutas de entrada a la API, el cliente podrá crear objetos, consultar, eliminar, o actualizar los datos. Esta arquitectura es actualmente la forma más popular y segura para el intercambio de datos entre las partes, pero nuestro software resultado debe ser un sistema de tiempo real, con cierta cantidad de usuarios en línea. Por lo tanto, las respuestas a los eventos generados por los usuarios, que pueden pasar en el sistema, deben ser inmediatas. Si bien se podría realizar mediante el uso de una API Rest, cada solicitud del protocolo HTTP agrega un retraso de tiempo, como el tiempo de establecimiento de conexión y el tiempo de transferencia de los paquetes. Por otro lado, los eventos que se pueden generar en el sistema, por lo general, no mantienen una estructura

compleja y los datos que se envían no suelen ser muy extensos. Por ende, el protocolo HTTP no nos conviene, por ejemplo, pensemos que se quiere modelar un chat, en él los usuarios envían y reciben mensajes, si lo hacemos vía HTTP nuestra aplicación resultará muy lenta. Dicho esto, necesitamos otra tecnología que nos asista a afrontar este problema. Para esto existen los WebSockets.

Un WebSocket es la tecnología que nos permite establecer una conexión entre el navegador del usuario y el servidor. Esta conexión es bidireccional y dura el tiempo que se mantenga abierto el navegador del cliente, o cuando se decida cerrar programáticamente. Cuando hablamos de bidireccional nos referimos a que ambos, tanto el cliente como el servidor, pueden enviar y recibir mensajes por el canal o conexión establecida. A diferencia del protocolo HTTP, se establece una única conexión TCP [28] en un puerto totalmente distinto, haciendo que sea mucho más rápido el intercambio de datos. Los WebSockets innovaron la web y provocaron que sea aún más dinámica, por esta razón se estandarizó y se generó un protocolo propio llamado WS, que al igual que HTTPS, también tiene su versión segura, WSS.

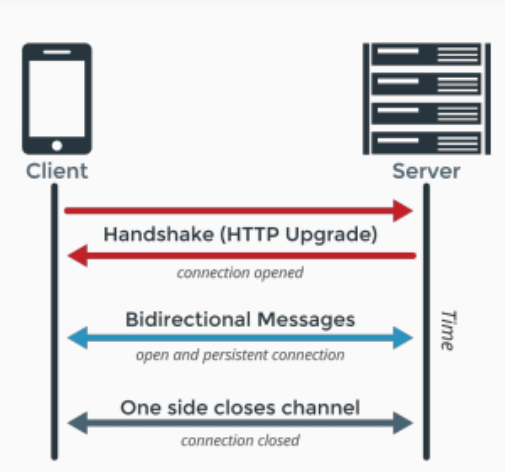

Figura 6.2: Comunicación de WebSocket.

En la figura 6.2, se puede observar el funcionamiento del protocolo de WebSockets, que incluye el intercambio de mensajes para establecer la conexión [29] (Handshake), mensajes con datos y el cierre de la conexión. Esta conexión puede parecerse a la del protocolo HTTP, pero no es así, suele confundirse porque se permite al servidor interpretar parte de la negociación como HTTP, y luego cambiar a WebSocket. No vamos a entrar en detalle en este tema, pero es importante conocerlo para tener una visión general de la tecnología, y entender su funcionamiento interno.

Los WebSockets se utilizarán para administrar los eventos generados por los Awareness, de

esta forma nos garantizamos que los usuarios vean al instante las acciones que generan los demás. Por esto, esta tecnología es la que estará más presente en los siguientes capítulos.

6.9.1. Ventajas

- **Mejor procesamiento:** Se disminuye el uso de la red, ya que se evita que se use HTTP, que contiene grandes paquetes de datos.
- **Rendimiento:** Se reduce el tiempo de latencia, porque se establecen menos conexiones, haciendo más eficiente el rendimiento del servidor.
- **Soportado por la mayoría de los navegadores:** Los navegadores como Chrome, Mozilla y Explorer fueron incorporando la API de WebSockets a medida que la tecnología avanzaba.

6.10. Acceleo

Acceleo [30] es una herramienta open-source creada por la Fundación Eclipse [31] que nos permite generar código y que implementa la especificación de modelo a texto de OMG [32]. Acceleo ayuda al desarrollador a manejar el ciclo de vida de sus generadores de código. Gracias a un enfoque basado en prototipos, puede crear de forma rápida y fácil su primer generador a partir del código fuente de un prototipo existente, luego, con todas las características de las herramientas Acceleo, como las herramientas de refactorización, mejorará fácilmente su implementación para realizar un generador de código completo.

Para realizar una transformación, Acceleo provee un lenguaje de programación basado en etiquetas, cumpliendo con las estructuras de condiciones básicas y la posibilidad de declarar variables mientras se recorre el modelo. Con estas etiquetas podemos crear plantillas, módulos y consultas (funciones específicas) que nos ayudan a recorrer nuestro modelo y generar los archivos necesarios con nuestro código.

En la siguiente figura 6.3, se presenta un ejemplo del lenguaje basado en etiquetas de Acceleo, en donde se crea un módulo que a su vez usa una plantilla (template) para crear una clase Java. En la figura 6.4, podemos ver el resultado que se obtiene cuando se procesa este módulo en

Acceleo. En este caso, se está generando un archivo Java que contiene una clase nombrada City, con sus variables de instancia y sus métodos. Así de sencillo es crear archivos con Acceleo.

generate.mtl

```
[comment encoding = UTF-8 /]
[module generate('http://www.eclipse.org/uml2/5.0.0/UML')]


[template public generateElement(aClass : Class)]
[comment @main/]
[file (aClass.classFileName(), false, 'UTF-8')]
package [aClass.containingPackages().name->sep('.')/];

  public class [aClass.name.toUpperFirst()/] {
  [for (p: Property | aClass.attribute) separator('\n')]
    private [p.type.name/] [p.name/];
  [/for]

  [for (p: Property | aClass.attribute) separator('\n')]
    public [p.type.name/] get[p.name.toUpperFirst()/]() {
      return this.[p.name/];
    }
  [/for]

  [for (o: Operation | aClass.ownedOperation) separator('\n')]
    public [o.type.name/] [o.name/]() {
      // TODO should be implemented
    }
  [/for]
  }
[/file]
[/template]
```

Figura 6.3: Ejemplo de módulo de Acceleo.

generate.mtl | City.java

```
public class City {
  private City city_at;

  private Street street_2;

  public City getCity_at() {
    return this.city_at;
  }

  public Street getStreet_2() {
    return this.street_2;
  }

  public Boolean isBig() {
    // TODO should be implemented
  }

  public Integer TestSequence() {
    // TODO should be implemented
  }
}
```

Figura 6.4: Resultado del módulo de Acceleo.

Usaremos Acceleo dentro del IDE [33] Eclipse, ya que es una extensión del mismo y aprovecharemos todas las ventajas que nos concede.

6.10.1. Ventajas

- **Sintaxis simple:** El lenguaje de etiquetas es de fácil aprendizaje y lectura.

- **Generación de código eficiente:** Otorga un buen rendimiento en la generación del código, una de las características que lo hace un generador de código de alta calidad.
- **Herramientas Avanzadas:** Dentro del IDE contiene su propia perspectiva con una barra de herramientas que facilita su uso.
- **Características a la par con el JDT [34]:** Integración con las herramientas de desarrollo de Java usadas por el IDE.

Capítulo 7

Transformación Técnica

7.1. Introducción

En este punto, ya conocemos la teoría de MDD y de cómo funcionan los sistemas colaborativos. Así como también, el funcionamiento del metamodelo, los conceptos y objetivos de cada uno de sus componentes, y las tecnologías. Estamos en condiciones de introducirnos en la transformación técnica de la herramienta desarrollada en este trabajo.

En este capítulo, veremos cómo se realiza la transformación técnica de un modelo creado con nuestro metamodelo de sistemas colaborativos. Para poder explicar este tipo de transformación iremos mencionando ejemplos, y anunciando cuáles fueron las decisiones que se tuvieron que tomar. Porque aunque el metamodelo nos otorgue los componentes y sus asociaciones, se tuvo que decidir algunos aspectos de funcionalidad para darle coherencia al sistema final, y poder cumplir con los objetivos conceptuales de cada componente.

7.2. Estrategia de la Transformación

Antes de comenzar, debemos dejar en claro cuál es el propósito de esta transformación técnica.

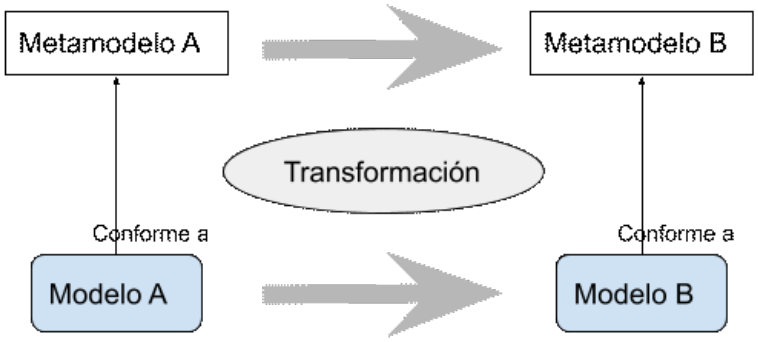

Figura 7.1: Arquitectura de Transformación.

La figura 7.1, nos presenta la arquitectura que se utiliza para aplicar una transformación. Para comprender esta arquitectura y la finalidad de nuestra transformación técnica, podemos contextualizar los elementos que se muestran en la figura. El "Metamodelo A" es el metamodelo de sistemas colaborativos, ya hemos visto, que con él se pueden crear una gran cantidad de modelos. En este caso, elegimos uno, por ejemplo tomamos el juego de ajedrez, que sería el "Modelo A". Esto se suele decir de la siguiente manera, el "Modelo A" es conforme a "Metamodelo A". El objetivo de esta transformación es obtener el "Modelo B", tomando como base el "Modelo A". Pueden existir múltiples herramientas que realicen transformaciones del "Metamodelo A", por ejemplo, a modelos de distintos lenguajes de programación. La herramienta que nosotros hemos desarrollado se encarga de transformar este "Modelo A" (modelo de Ajedrez), a un "Modelo B", cuya base es el lenguaje Typescript, y además, se utilizan las tecnologías mencionadas previamente.

La transformación técnica de todo el modelo completo, se lleva a cabo a través de módulos que nos aporta Acceleo. Estos contienen la lógica que se encarga de recorrer el modelo y procesar cada uno de los elementos, para crear los archivos necesarios del cliente y del servidor, y lograr obtener un sistema web funcional, en base a nuestro modelo. A lo largo de este capítulo expondremos figuras que contienen porciones de código de estos módulos, sobre todo de secciones específicas del tema que se abarque, ya que suelen ser módulos muy extensos para presentar en este documento. Si se desea tener una visión total se recomienda revisar los archivos directamente desde el repositorio de la herramienta.

7.3. Transformación Técnica de los Elementos del Metamodelo

En esta sección hablaremos de las transformaciones de cada uno de los elementos del metamodelo, enseñando los componentes que se generan para cada tecnología y explicando el comportamiento, en base a las decisiones que se fueron tomando durante el transcurso del desarrollo. Recordemos que existe funcionalidad que depende del sistema final, que se desea crear en base a un modelo, y que escapan del alcance del metamodelo. Por este motivo, existen soluciones, para estas funcionalidades específicas, que se desarrollaron con fines demostrativos, y también, soluciones que creemos que son las más adecuadas para el comportamiento de un sistema colaborativo.

En el mismo orden, como lo realizamos en capítulos previos, iremos mencionando la transformación técnica de los elementos, entrando en detalle en las secciones de cada uno de ellos. A lo largo de cada sección, se presentan los componentes y clases que se generan, sus comportamientos y también la ubicación dentro del sistema resultante.

La herramienta contiene una estructura de directorios por defecto, basada en los estándares de cada tecnología. No se profundizará el tema en este punto, solo se nombraran algunos de sus directorios. No obstante, en el "Anexo I – Estructura de la Herramienta" se puede ver una breve explicación de las carpetas y archivos que forman parte de esta estructura.

7.3.1. Transformación Técnica de Operaciones

Recordemos que las operaciones son las acciones que el usuario puede realizar en el sistema web resultante. En el modelo se incluyen dentro de los elementos colaborativos, como los espacios de trabajo, las actividades y las herramientas. Existen dos tipos de operaciones que brinda la herramienta, su clasificación se debe según su comportamiento. Para el primer tipo la herramienta solo se encarga de su visualización, no brinda ningún comportamiento extra, este depende del desarrollador. Es decir que la herramienta crea los archivos necesarios para que el desarrollador complete su lógica. Para el segundo tipo, además de la visualización, la herramienta asigna un comportamiento por defecto. Este tipo de operaciones está vinculado fuertemente con la ejecución de los procesos.

En la transformación conceptual, hemos mencionado que toda operación es un botón, ya que es el mejor camino para representar una acción. Pero dentro de la transformación técnica, no son solo botones en un archivo HTML. Cuando se presentó la tecnología Angular se comentó que está orientada en componentes, es decir que todo lo que se quiera representar es un componente. Estos componentes están compuestos por tres archivos, un HTML que contiene la vista del componente, un archivo TypeScript que funciona como controlador, que incluye la clase que lo representa y donde se mantendrá la lógica propia del componente y por último un archivos CSS para los estilos del mismo. Todas nuestras operaciones tendrán su propio componente. Algunos se generarán sin lógica, pero con la estructura adecuada, para que el desarrollador extienda su comportamiento como lo desee. Otros componentes tendrán una lógica por defecto, propia de la herramienta. Esto no quiere decir que el desarrollador no pueda extenderlos, en realidad podría modificar cualquier componente de la herramienta sin problemas, solo que deberá tener cuidado para no perjudicar su correspondiente funcionamiento.

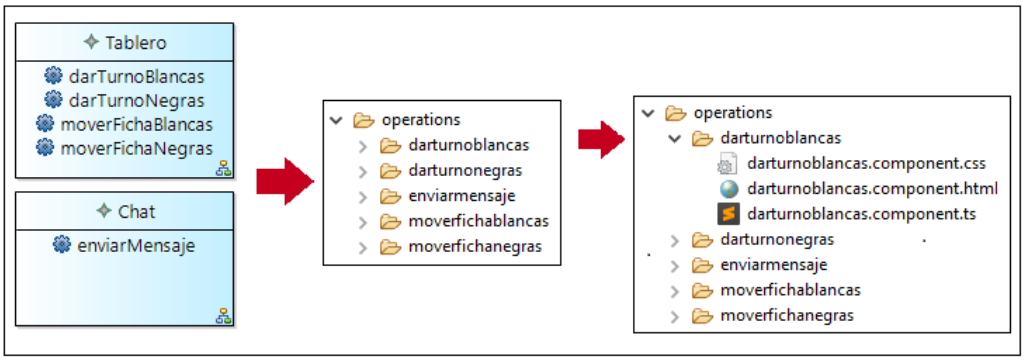

Figura 7.2: Transformación técnica de operaciones.

En la figura 7.2, de izquierda a derecha, podemos observar la transformación de las operaciones. Dentro de la aplicación del cliente (Angular), todos los componentes de las operaciones se crearán en el directorio "operations". En la figura, también podemos ver, un ejemplo de un componente "darTurnoBlancas", creado por la herramienta, para esta tecnología. Esta operación podría asociarse a un rol llamado "Juez", que sería el encargado de distribuir los turnos dentro de la partida de ajedrez. De esta forma, esta operación sólo podría ser visible por este rol. Las operaciones "moverFichaBlancas" y "moverFichaNegras", podrían ser acciones que solo pueden realizar los jugadores. La operación "enviarMensaje" se podría asociar a los roles de tipo "Jugador" como así también a los de tipo "Espectador". Es así como se podría modelar una partida de ajedrez con nuestro metamodelo, aunque pueden existir otras variantes. Todas

las operaciones del ejemplo se visualizarán dentro de la herramienta que le corresponde, en este caso “Tablero” y “Chat”.

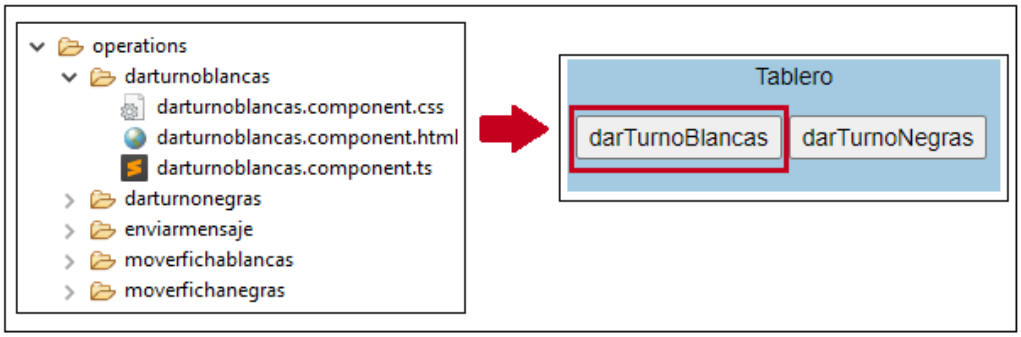

Figura 7.3: Transformación técnica de operación “darTurnoBlancas”.

En la figura 7.3, se muestra como se visualiza el componente “darTurnoBlancas” en la vista. Cuando el usuario accione el botón marcado, comenzará a ejecutar su lógica, según lo modelado.

En la herramienta de transformación, un módulo de Acceleo, se encarga de crear este componente para la vista, como se puede observar en la figura 7.4. Este módulo se encarga de invocar a tres nuevos módulos, para crear los archivos que corresponden, en este caso, a un componente Angular, es decir, los archivos HTML, TypeScript y CSS. Estos módulos usan el template de tipo file de Acceleo, pero no se presentan en este documento, ya que son muy extensos, se recomienda examinarlos en el repositorio que contiene el código de la herramienta.

```
[**
 * Genera un componente de una operacion para la vista.
 */]
[module generate_operation('http://cm/1.0')]

[import org::lifia::collaborativeTool::files::operation::generate_operation_component/]
[import org::lifia::collaborativeTool::files::operation::generate_operation_view/]
[import org::lifia::collaborativeTool::files::operation::generate_operation_style/]

[template public generateOperation(aCollaborativeModel : CollaborativeModel, aOperationName: String)]
[generateOperationComponent(aCollaborativeModel, aOperationName)/]
[generateOperationView(aCollaborativeModel, aOperationName)/]
[generateOperationStyle(aCollaborativeModel, aOperationName)/]
[/template]
```

Figura 7.4: Módulo Acceleo encargado de crear los componentes para las operaciones.

En el metamodelo, los elementos colaborativos como los espacios de trabajo, las actividades y las herramientas pueden incluir operaciones, que se agregan mediante los editores. Cuando se ejecutan los módulo de cada uno de ellos, se itera sobre las operaciones para invocar el módulo de la figura 7.4. La iteración se realiza sobre una colección de operaciones, llamada

“elementOperation”, que provee el metamodelo para cada componente colaborativo. En la figura 7.5 se muestra un ejemplo de cómo se realiza una de estas iteraciones. Por cada iteración, se crea una instancia de la clase ElementOperation. luego una instancia de la clase Operation, que se encuentra embebida en el atributo “operation”, y por último, se obtiene el nombre de la operación para invocar al módulo propiamente dicho. Este es un pequeño ejemplo de cómo se elaboraron las iteraciones en la herramienta, algunas son mucho más complejas que otras.

```
[for (aWorkspaceElementOperation : ElementOperation | aWorkspace.elementOperation)]
    [let aWorkspaceOperation : Operation = aWorkspaceElementOperation.operation]
        [generateOperation(aCollaborativeModel, aWorkspaceOperation.name)/]
    [/let]
[/for]
```

Figura 7.5: Ejemplo de iteración de operaciones en un workspace.

Inicialmente, se indicó que existen dos tipos de operaciones, las que se generan sin comportamiento, y las que tienen un comportamiento por defecto. Las primeras son solo parte del cliente, depende del desarrollador extender su funcionalidad. Las segundas también tienen su pertinente componente en el cliente, pero a su vez, forman parte del servidor. Como son operaciones que se relacionan directamente con la ejecución de los procesos, se asocian con archivos de configuraciones de las actividades. A su vez, tienen su clase representativa, pero como siempre se comparten los mismos atributos, independiente del modelo, esta clase es genérica para cualquier operación. Esto quiere decir, que si todas las operaciones que se pueden generar con el metamodelo están compuestas por nombre y rol, no tiene ningún sentido crear por cada modelo una clase particular para una operación. Por esta razón, se decidió que las operaciones dentro del servidor serán representadas por una clase genérica. El comportamiento de estas operaciones será dado por configuraciones de las actividades.

Cuando las operaciones se asocian a las actividades, las operaciones comienzan a cumplir su cometido, que es poder permitir a los usuarios interactuar con el sistema y navegar a través de las diferentes actividades. Para ser más específicos, esta asociación se lleva a cabo a través del protocolo de las actividades, que entraremos en detalle, cuando expliquemos el funcionamiento de las actividades y sus protocolos

En resumen, estos elementos se representan mediante botones y funcionan como acciones por parte del usuario, que pueden o no tener comportamiento dependiendo de sus asociaciones. Pueden ser visibles o no, según los roles vinculados y pueden definir el rumbo de la navegación de

las actividades de los procesos. Sea cual sea el tipo de operación modelada, su comportamiento puede ser extensible y no necesariamente deben ser representadas por un botón, son decisiones que el desarrollador deberá tomar.

Y para finalizar, cómo lo haremos con todos los elementos del modelo que enseñemos en este capítulo, haremos una síntesis general a través de una tabla, que nos ayuda a plasmar su descripción, su transformación conceptual y la transformación técnica en cada una de las aplicaciones resultantes.

Elemento	**Descripción**	**Transformación Conceptual**	**Aplicación Cliente**	**Aplicación Servidor**
Operación	Acciones que realiza el usuario.	Botón	Componente Angular	Archivos de configuración

Tabla 7.1: Tabla de transformación de operaciones.

7.3.2. Transformación Técnica de Espacios de Trabajo

Los espacios de trabajo o workspaces son los encargados de darle un marco a la ejecución de las actividades, y a su vez pueden contener sus propias operaciones.

Luego de aplicar la transformación, a nivel cliente y al igual que las operaciones, se crean componentes Angular, y se guardarán en su debido directorio ("workspace"), como se muestra en la siguiente figura.

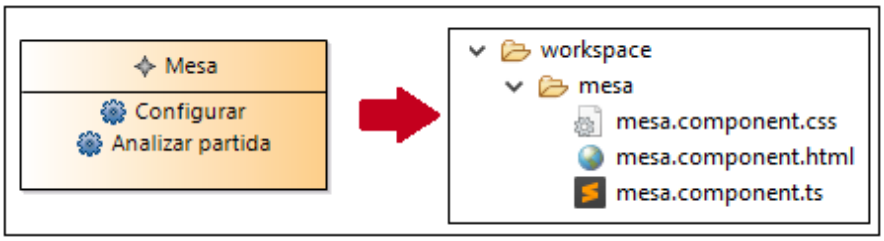

Figura 7.6: Transformación técnica de workspace.

Cuando la herramienta de transformación procesa un workspace, mediante su módulo de Acceleo, se recorren sus operaciones y se invoca el módulo, que se encarga de crearlas, visto previamente.

Para el usuario, los espacios de trabajo y las actividades van juntos. Es decir, que cuando se cargue una actividad, se podrá observar el workspace que la contiene. De esta forma, cuando el usuario ingrese a una instancia de un proceso, se cargará la primera actividad o la corriente, y se visualizará su workspace donde se realiza esa actividad. Esto se puede traducir, diciendo que el workspace es el contenedor padre de la actividad, como se ejemplifico en la figura 5.2 del Capítulo 5 ("Transformación Conceptual").

Por otro lado, Angular suministra rutas para acceder a algunos componentes y permitirnos la navegación del sitio. Basta con poner esta ruta en el navegador para acceder a su componente asociado. A medida que los usuarios van interactuando con la aplicación web, efectúan transiciones entre las distintas actividades de los procesos, y se realizan mediante nuestras rutas. Los workspaces forman parte de estas rutas, porque no solo nos posibilitan el acceso al componente sino que también nos ayudan a identificar la relación con la actividad. Por ejemplo, si nosotros modelamos un workspace llamado "Aula1" que contiene una actividad llamada "RendirFinal", para acceder al componente que representa esta actividad dentro de "Aula1", la herramienta generó la siguiente ruta "/aula1/rendirfinal". Si ingresamos a ella, vamos a ver cargado todo el componente que representa esta relación. Por otro lado, puede ocurrir que existan en otros procesos la misma actividad "RendirFinal" en otro workspace llamado "Aula2". En este caso, la ruta cambiaría a "/aula2/rendirfinal" y la relación es otra. Nuestras rutas nos permiten asociar e identificar las relaciones que existen entre los workspaces y las actividades.

En el servidor, no se obtuvo motivo alguno para crear una clase que represente a los workspaces, en este caso la actividad es la que debe saber en qué contexto o espacio de trabajo se ejecuta. Por lo cual los workspaces se convierten en un atributo de la clase que representa la actividad. Para determinar esta asociación entre un workspace y una actividad, la herramienta procesa las relaciones del metamodelo llamadas BelongsRelationship, o relaciones de pertenencia. Cuando comienza la transformación, los módulos de Acceleo procesan este tipo de relación y crean para cada una de ellas, los archivos necesarios para el workspace y la actividad.

Para concluir, presentamos su tabla, que nos resume la transformación global de los workspaces.

Elemento	Descripción	Transformación Conceptual	Aplicación Cliente	Aplicación Servidor
Espacio de Trabajo.	Entorno donde se desarrollan las actividades del usuario.	Contenedor de actividades y operaciones.	Componente Angular con endpoints de navegación	Atributo asociado a la actividad.

Tabla 7.2: Tabla de transformación de workspaces.

7.3.3. Transformación Técnica de Actividades Colaborativas

Las actividades son las tareas que puede realizar el usuario dentro de un proceso, y están contenidas por los workspaces. En ellas a su vez se pueden encontrar operaciones y las herramientas que el usuario utiliza. Nuevamente al igual que los elementos ya vistos, las actividades se convierten en componentes Angular que se guardan en el directorio "activity" dentro del proyecto del cliente. Estos componentes también tienen sus rutas de navegación.

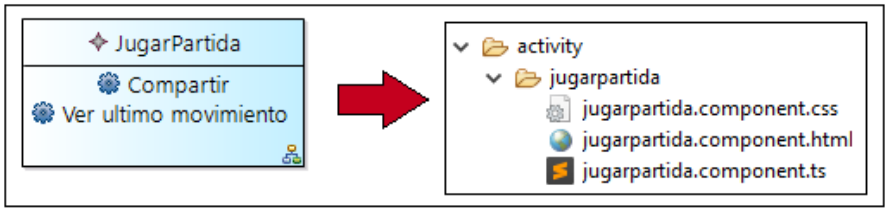

Figura 7.7: Transformación técnica de actividad.

Del lado del servidor, las actividades tienen su propia clase en un archivo TypeScript, nombrada "ActivityContext". Esta clase representa el modelo de cada actividad, y el esquema en la base de datos. Sus instancias son creadas, modificadas y eliminadas por un controlador de actividades, que se encarga de manipularlas y comunicarse con diferentes servicios para el funcionamiento de las mismas. Los puntos de acceso al controlador son métodos asociados a endpoints que permiten al cliente comunicarse con el servidor, e indicar qué operación se desea realizar.

Para este desarrollo, no se encontró ningún motivo para crear por cada actividad modelada una clase específica. Las actividades que se pueden crear en el metamodelo comparten los

mismos atributos, por lo cual pueden ser representadas con una misma clase. La distinción entre las actividades se realiza a través de sus configuraciones y sus protocolos, que asimismo tienen sus servicios genéricos correspondientes.

En la figura 5.3 y 5.4, del Capítulo 5 ("Transformación Conceptual"), pudimos ver cómo se visualizará una actividad luego de aplicar la transformación, pero no entramos en detalle con respecto al funcionamiento y su ejecución. El responsable de la visualización de una actividad es el componente Angular que se comentó al principio de este apartado. Por otro lado, la ejecución de la actividad, está dada por la interacción del usuario con este componente. El mismo se comunica con el servidor, vía API Rest, dejando la ejecución de la actividad en manos del controlador y los servicios nombrados previamente. Ellos no solo se encargan de gestionar las instancias de ejecución de las actividades, sino que también, manejan la sincronización de los usuarios que participan en una ejecución. Recordemos que cuando se está ejecutando una actividad, los usuarios pueden ingresar, realizar operaciones, utilizar las herramientas y salir cuando lo deseen.

Cuando veamos en profundidad el funcionamiento de los procesos, sabremos el momento en el que se crean las instancias de actividades. Para facilitar la comprensión, en esta sección asumimos que dos usuarios quieren jugar una partida de nuestro ajedrez, ingresan al sistema y seleccionan la opción "JugarPartida". Veamos a continuación, el funcionamiento de una actividad para esta situación. Primero, modelamos nuestra actividad "JugarPartida", le asociamos un workspace, para que se visualice en un contenedor, llamado "Mesa", y le vinculamos dos herramientas que los usuarios podrán utilizar en ella, una herramienta "Tablero", simulando nuestro tablero físico de ajedrez, y un "Chat" para que se puedan comunicar. Por último, definimos qué usuarios pueden participar en nuestra actividad, para realizar esto, creamos el rol "Jugador". En estos momentos, tenemos nuestro modelo de la siguiente forma.

La figura nos muestra la actividad en color gris, las herramientas en color verde, el rol de colaboración en color celeste y por último nuestro workspace en color naranja. Podemos observar que las flechas nos indican que estos elementos se vinculan con nuestra actividad. Estas asociaciones se representan técnicamente con las relaciones que otorga el metamodelo, y cada una de ellas tienen, dentro del mismo, una clase que las representa. La clase ParticipationRelationship (Relación de Participación), relaciona la actividad con los roles de colaboración. La clase UseRelationship (Relación de Uso), relaciona la actividad con las herramientas. Por últi-

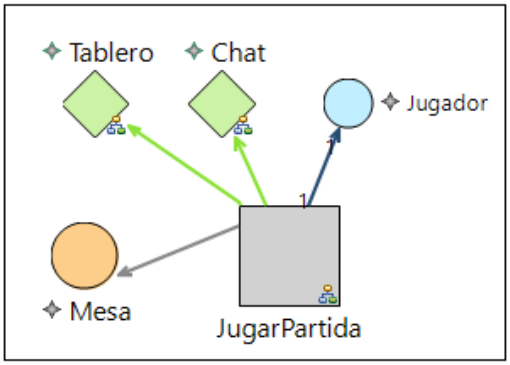

Figura 7.8: Modelo básico de la actividad “JugarPartida”.

mo, BelongsRelationship (Relación de Pertenencia), relaciona la actividad con los workspaces. Todas ellas se tienen en cuenta durante el recorrido del modelo, en el proceso de transformación, y se crean los componentes de los elementos en las correspondientes tecnologías.

Si en estos momentos, aplicamos la transformación y luego levantamos nuestro sistema web resultante, los usuarios de rol “Jugador” que se registren e inicien sesión, cuando ejecuten la actividad “JugarPartida”, la verán de la siguiente manera.

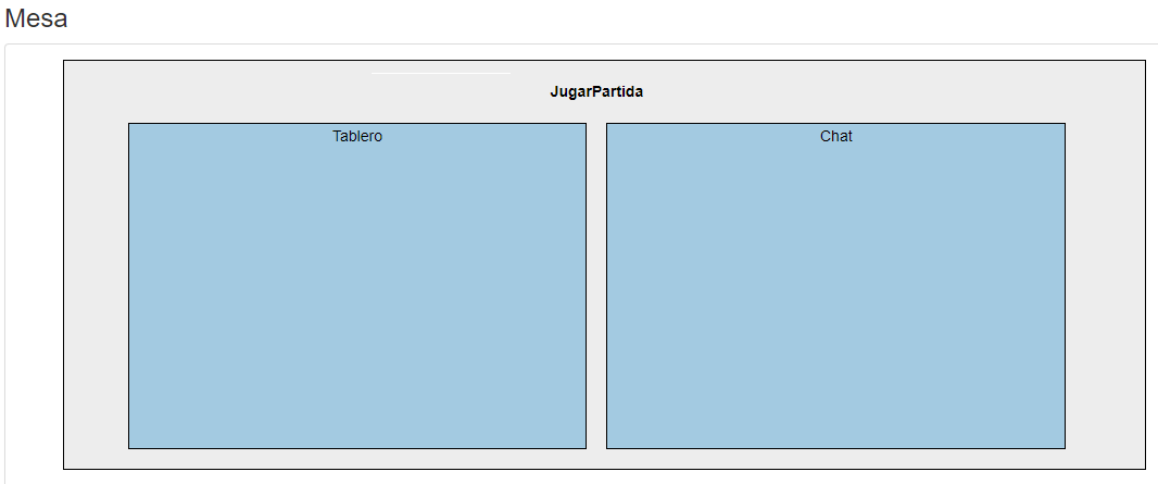

Figura 7.9: Actividad “JugarPartida”, sin operaciones, luego de aplicarle la transformación.

Aún no hemos asignado ninguna operación, solo nos garantizamos que los usuarios puedan iniciar y ver la actividad.

Hasta aquí, únicamente tenemos dos jugadores, y nuestro comportamiento deseado es, que el sistema resultante sea lo suficientemente inteligente para sincronizarlos, y comenzar a ejecutar la partida. En breve veremos cómo se abordó este punto, pero antes debemos comentar una problemática que surgió durante el desarrollo. Si nos ponemos un poco más detallistas en

nuestro ejemplo, sería muy bueno que al ingresar un jugador, se le asigne una distinción que lo identifique, por ejemplo el color de las piezas que habitualmente se usan en una partida de ajedrez. Es decir, que un jugador se podría representar con el color "Negro" y el otro con el color "Blanco". También se podrían conceder acciones a determinado color, aunque para nuestro caso no es válido, ya que queremos que nuestros jugadores se comporten de la misma manera. Todo indica que al ingresar a la actividad los jugadores cambiarán "automáticamente" a un nuevo rol, donde pueden encontrarse con posibles nuevas acciones, según lo modelado. El metamodelo ofrece la posibilidad de cumplir con este requerimiento, y de una forma muy sencilla, solo hace falta modelar dos nuevos roles "Negras" y "Blancas", haciendo referencia al color de piezas que se usan en un ajedrez, y vincularlos con la actividad.

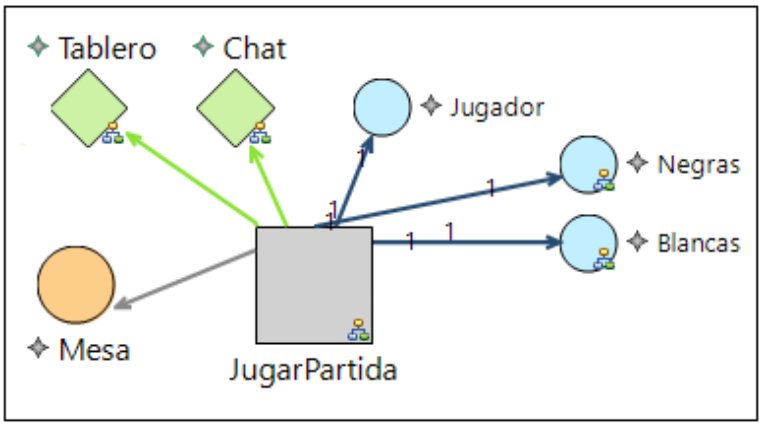

Figura 7.10: Modelo de actividad "JugarPartida" con roles "Negras" y "Blancas".

Si analizamos la situación, surge un nuevo inconveniente, como le decimos a nuestra herramienta que los roles "Negras" y "Blancas" se asignan automáticamente una vez que el usuario ingresa a la actividad. Esta consideración no está contemplada ni es posible desde el modelo, ya que no es capaz de discriminar los tipos de roles que se pueden llegar a modelar. El objetivo del metamodelo es ayudarnos a crear cualquier tipo de sistema colaborativo, y permitirnos conformar infinidades de modelos, no sabemos el alcance de cada modelo. Esta cuestión depende de las reglas de negocio y los requerimientos del desarrollador que utiliza la herramienta. Para solventar el problema, se decidió delegar la responsabilidad al desarrollador, a través de configuraciones, que deberán completarse una vez aplicada la transformación.

La configuración de la que hablamos en la sección anterior no es la única que depende del desarrollador. Si volvemos a nuestro ejemplo, tener solo dos jugadores, uno para las piezas

negras y otro para las piezas blancas, es suficiente, y cumplen con nuestro propósito. Pero solo es válido para este tipo de juegos, en donde solo participan como máximo dos jugadores. Qué ocurriría si se quiere modelar un juego donde participan dos equipos, pero cada uno de ellos está conformado por cinco jugadores. Esta es una situación completamente correcta y modelable. En este caso, la herramienta debe ser capaz de sincronizar, por ejemplo, cinco jugadores para el rol "Azul" y cinco jugadores para el rol "Rojo". Nuevamente, esta cantidad de usuarios, que pueden participar bajo el nombre de un rol en una actividad, es totalmente configurable, y está sujeta al alcance del modelo.

Las tecnologías que corren bajo nuestra aplicación del cliente y del servidor, están basadas en NodeJS. En este entorno es habitual, en el momento de incorporar o utilizar algún middleware, realizar configuraciones en archivos JSON. Se ha aprovechado la facilidad de creación de estos archivos para presentarle a los desarrolladores estas configuraciones, además de ser una tecnología bastante conocida y comprensiva por los mismos. Teniendo presente este tema, una vez aplicada la transformación, el desarrollador podrá configurar la cantidad de usuarios por rol que pueden participar en una actividad. La herramienta se encarga de crear un archivo de configuración, en el directorio "/config" dentro de la aplicación del servidor, llamado "activity-role.configuration", con la siguiente estructura.

```
'jugarpartida': {
            'roles': [
                    {
                        'name': 'blancas',
                        'onlineUsersNumberAllowed': 1,
                        'allowAdvanceRoles': false
                    },
                    {
                        'name': 'negras',
                        'onlineUsersNumberAllowed': 1,
                        'allowAdvanceRoles': false
                    },
            ]
},
```

Figura 7.11: Estructura de archivo de configuración "activity-role.configuration" para ejemplo de la actividad "JugarPartida".

En él se puede configurar, a través del atributo nombrado "onlineUsersNumberAllowed", la cantidad de usuarios que participan por cada tipo de actividad y por cada rol. De esta forma, le decimos a nuestro sistema resultante que sincronice cierta cantidad de usuarios, de

un determinado rol, para una actividad. El atributo “allowAdvanceRoles” también tiene su impacto en la ejecución de las actividades, pero se explicará en la sección de procesos de este capítulo. La figura solo muestra una sección de la estructura, que es la actividad de nuestro ejemplo, pero también, en el caso de tener más actividades, se encontrarán listadas en la misma, y con sus respectivos roles

El sistema web generado se basa en estas configuraciones de las actividades, para sincronizar los usuarios. Sincronizar los usuarios significa gestionar la cantidad de usuarios en línea para determinar cuando la actividad puede comenzar a ejecutarse. Para ello, el sistema emplea un conjunto de estados internos, en el que van pasando las actividades según esta cantidad. En este trabajo se ha investigado el manejo de los usuarios en distintos sistemas colaborativos, para definir cuál es la mejor solución con respecto a este tema. Y se optó por hacer esperar a los usuarios hasta que llegue la cantidad configurada según los roles. Esto quiere decir, que el sistema realiza una serie de cálculos, en base a estas configuraciones, para establecer que la actividad puede ejecutarse. Si los usuarios conectados no llegan a cumplir con estos cálculos, la actividad se considera que está en un estado pendiente (“pending”). Y el usuario solo verá una etiqueta, como lo muestra la siguiente figura, que indica que se está esperando que ingresen nuevos usuarios.

Figura 7.12: Actividad en estado ‘pending’.

Esto no significa que el usuario no pueda seguir usando el sistema, sino que solo para esa actividad específica, se necesita que se incorporen más participantes. Esta etiqueta se agregó para indicar al desarrollador cómo se comporta el sistema cuando no están todos los usuarios disponibles, y se puede modificar a su gusto, según sus objetivos.

Si volvemos a nuestro ejemplo de la actividad “JugarPartida”, y la configuramos para que solo entren dos usuarios, uno de rol “Blancas” y otro de rol “Negras”. El primer usuario que entre a ejecutar la actividad, verá la etiqueta previa de la figura 7.12. Cuando el segundo usuario ingrese, ya se cumplirá la configuración indicada, y la actividad comenzará a ejecutarse, estableciendo un nuevo estado interno, llamado “running”. Es aquí en donde se habilitan todas las operaciones

que los participantes pueden realizar según su rol. Al igual que el estado “pending”, y con fines demostrativo, se le proporciona al desarrollador una visualización del estado como parte del título de la actividad, que no necesariamente debe ser visible en el sistema final.

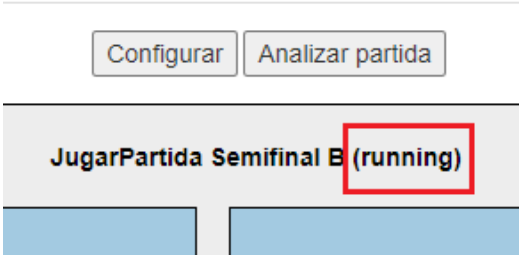

Figura 7.13: Actividad en estado “running”.

Es importante aclarar, que una vez que la actividad esté en estado “running”, no se puede volver al estado “pending”. Cuando la actividad comienza a ejecutarse, no se puede detener, y solamente finalizará bajo ciertas condiciones que determinan los procesos colaborativos, vistos en la próxima sección. Cuando ocurren estas condiciones, la actividad finaliza, y se considera que entra en un estado interno llamado “finished”.

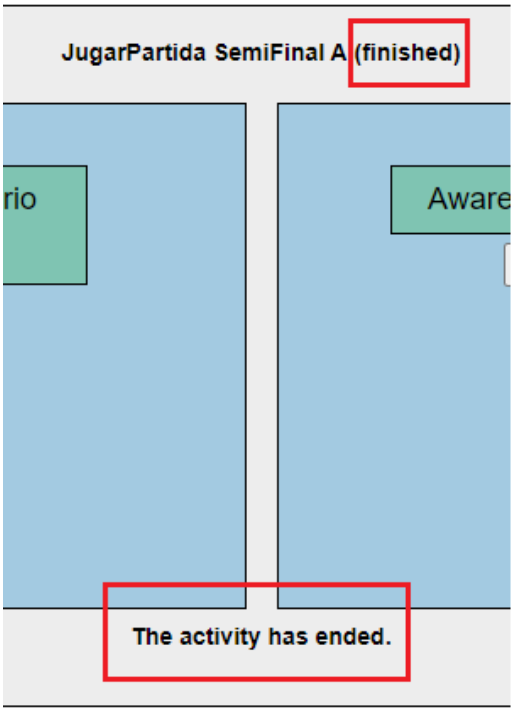

Figura 7.14: Actividad en estado “finished”.

Visualmente el estado finalizado se muestra igual que el estado en ejecución, es decir en el título de la actividad, pero a su vez se hace visible una leyenda en la parte interior de

la misma. Todas las operaciones asociadas a la ejecución de la actividad desaparecen cuando finaliza. La actividad sigue existiendo, para futuras consultas, pero también es responsabilidad del desarrollador mantener este comportamiento o extenderlo a su manera.

Como resumen del proceso de transformación de las actividades colaborativas, presentamos la tabla al igual que las secciones anteriores.

Elemento	**Descripción**	**Transformación Conceptual**	**Aplicación Cliente**	**Aplicación Servidor**
Actividad Colaborativa	Tareas que realiza el usuario durante un proceso.	Contenedor de herramientas y operaciones con sincronización de usuarios, y estados.	Componente Angular. Ruta de navegabilidad.	Ruta de API Rest. Esquema de Base de Datos. Controlador y configuración.

Tabla 7.3: Tabla de transformación de actividades.

7.3.4. Transformación Técnica de Protocolos de Estados

Lo visto en la sección previa, son los estados internos que usa el sistema para administrar la sincronización de los usuarios en cada actividad, pero por otro lado, el metamodelo permite modelar actividades con estados. Durante la ejecución una actividad puede ir cambiando de estado según las interacciones que tengan los participantes, a esto se lo conoce como protocolo de estado de la actividad. En el Capítulo 4, donde realizamos una pequeña descripción de cada elemento del metamodelo, se puede observar en la figura 4.6 un ejemplo gráfico de como se ve un protocolo de estados. Para crearlo, el modelador debe ingresar al editor de protocolos de la actividad e ir creando los estados que desee. Luego, a cada estado se le puede asignar un conjunto de operaciones para cada rol y se puede elegir cuales de estas son operaciones de transición, es decir, que realizan un cambio de estado. De esta forma, mientras se ejecuta la actividad los estados irán cambiando hasta que se llegue a una operación que dé por finalizada la ejecución, dando paso a una nueva actividad o finalizando el proceso al que pertenece.

El protocolo de una actividad está representado dentro del metamodelo por las clases “CollaborativeActivityState” y “CollaborativeProtocolTransition” ambas se relacionan con la clase

“RoleElementOperation”. Y particularmente la clase “CollaborativeActivityState” mantiene una relación uno a uno con la clase que representa la actividad “CollaborativeActivity”.

A nuestra actividad “JugarPartida” podríamos agregarle dos estados, uno llamado “MoviendoBlancas” y otro “MoviendoNegras”, simulando que se están moviendo las fichas, y para que ambos jugadores sepan quien tiene el turno. Es decir, queremos que se comporte de la siguiente manera, cuando le toque el turno al rol “Blancas”, queremos que la actividad pase al estado “MoviendoBlancas” y solo se habiliten las operaciones, relacionadas a nuestro tablero, para el rol “Blancas”. Del otro lado, el rol “Negras” no verá las operaciones del tablero, pero si el resto de ellas, por ejemplo “enviarMensaje” en el chat. Para realizar esto, debemos dentro del tablero, modelar dos operaciones que nos permitan realizar la transición entre los estados. Recordemos que las operaciones son acciones o eventos por parte del usuario, en este caso, queremos “imitar” que se está moviendo físicamente una ficha, de modo que las nombraremos “moverFichasBlancas” y “moverFichasNegras”. Cuando el usuario de color blanco ejecute esta acción automáticamente la actividad cambiará su estado a “MoviendoNegras”, habilitando el mismo comportamiento pero para este rol. En la figura 7.15, se presenta el protocolo de estados descrito.

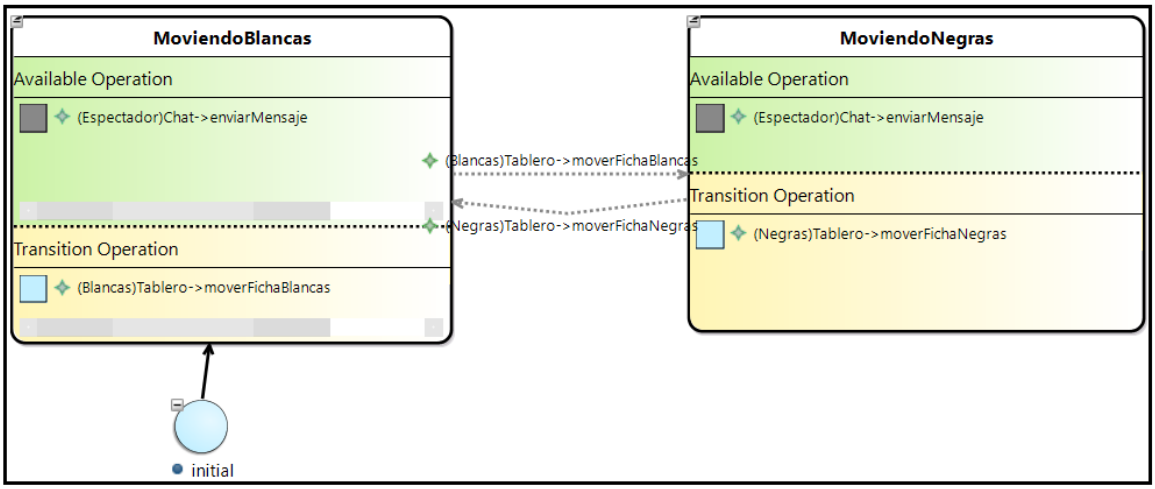

Figura 7.15: Protocolo de Estados para la actividad “JugarPartida”.

En la imagen previa se puede observar un círculo celeste con la leyenda “initial”, forma parte del modelado y nos permite indicar que estado es el inicial, en este caso cuando comience la ejecución de la actividad, se empezará por el estado “MoviendoBlancas”, es decir que el primer jugador que pueda mover una ficha será el del color blanco.

Desde el punto de vista técnico, la herramienta se basa en configuraciones para establecer

cuál es el próximo estado luego de cada transición. En la figura 7.15, dentro de cada estado podemos observar en un recuadro verde ("Available Operation") las operaciones que siempre podrá accionar el usuario cuando esté en ese determinado estado, pero estas operaciones no tendrán impacto en la ejecución. En cambio, las operaciones que se encuentran dentro del recuadro amarillo "Transition Operation", son las operaciones de transición modeladas y se encargan de cambiar de estado cuando se accionen.

En la interfaz, el usuario siempre verá el estado del protocolo a la derecha del nombre de la actividad, como se muestra en la figura 7.16, en el recuadro rojo de arriba a la derecha.

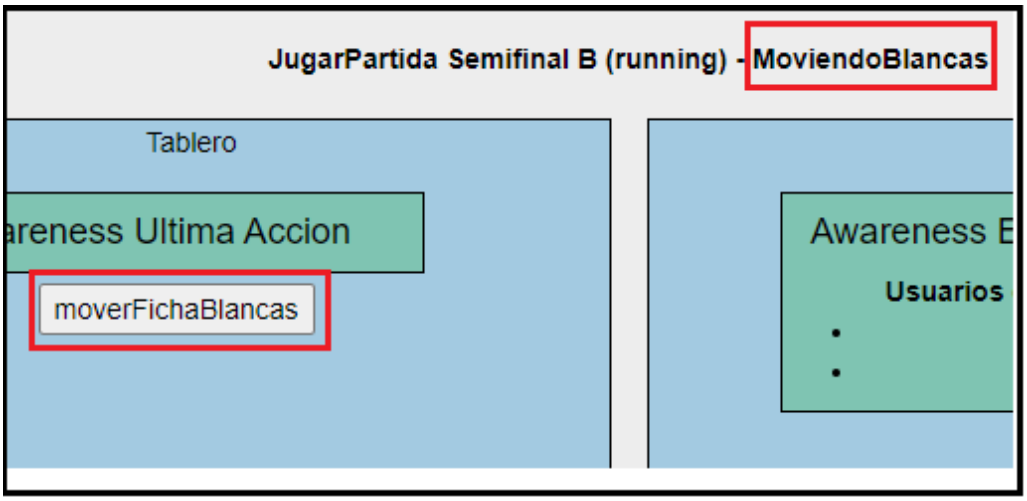

Figura 7.16: Interfaz de protocolo de estados para la actividad "JugarPartida".

En la figura 7.16, también se remarca la operación modelada en el protocolo. Cuando el usuario blanco presione este botón, el estado cambiará a "MoviendoNegras", y se le dará el turno al jugador de color negro. Y así, los jugadores moverán fichas hasta finalizar la partida, y la actividad irá cambiando de estado en el transcurso de las interacciones.

Los protocolos de las actividades nos ayudan a modelar una infinidad de situaciones en la ejecución de las actividades. Por ejemplo, se le podría agregar a nuestra actividad un nuevo estado "ChequeandoJugada", asociada a un nuevo rol llamado "Juez" que se encargue de comprobar la jugada de cada participante. Y en el caso que la jugada sea un "jaque mate" dé por finalizada la partida, notificando al ganador.

Por último, presentamos la tabla que corresponde a los protocolos de estados.

Elemento	Descripción	Transformación Conceptual	Aplicación Cliente	Aplicación Servidor
Protocolo de Estados	Estados por los que puede pasar una actividad en ejecución.	Secuencia de estados de transición de una actividad.	Estructura que acompaña a las actividades.	Archivos de configuración. Estructura que acompaña a las actividades. Servicio encargado de manejarlos.

Tabla 7.4: Tabla de transformación de protocolo de estados.

7.3.5. Transformación Técnica de Herramientas

Todas las operaciones que vimos en nuestro ejemplo, a lo largo de este capítulo, se encuentran en las herramientas modeladas. Esta es una decisión que se tuvo que tomar para este trabajo para limitar el uso de las operaciones en el sistema. Si bien se permiten modelar operaciones en las actividades y otros elementos del metamodelo, en el ejecución las operaciones de transiciones entre los estados se encuentran en el interior de las herramientas. Se optó por este camino ya que son los elementos que los usuarios usan para cumplir sus objetivos. Por ejemplo, si se modela una clase virtual, el profesor podría usar como herramienta una pizarra para enseñar a sus alumnos, es decir, realizaría su trabajo a través de la herramienta. Para el ejemplo del ajedrez, el objetivo por parte de los usuarios de jugar una partida, se realiza en el tablero, que es donde los usuarios mueven las fichas, y a su vez emplean el chat para comunicarse.

Al ser elementos que se visualizan en la interfaz y requieren interacción con los usuarios, para las herramientas también se generan elementos Angular, que contienen la lógica necesaria para el funcionamiento. Como elemento que contiene operaciones, luego de generar su correspondiente componente, genera los componentes de sus operaciones.

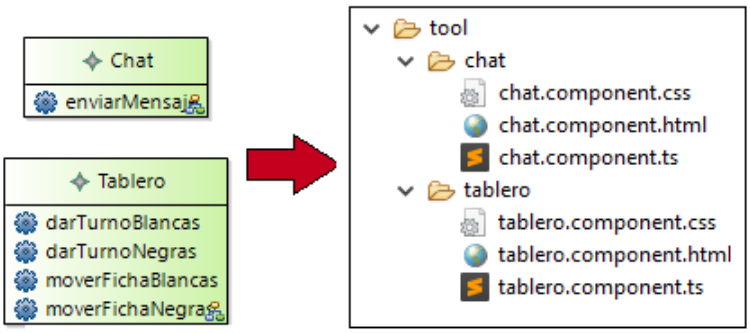

Figura 7.17: Transformación técnica de herramientas.

La clase que las representa se llama “Tool” y hereda de “CollaborativeElement”, que mantiene una relación con la clase de las operaciones “ElementOperation”.

Las herramientas tienen una relación de uso (“UseRelationship”) con las actividades, esto significa que las actividades hacen uso de las herramientas vinculadas, por esta razón se definió que las herramientas deben visualizarse dentro de las actividades, como se presentó en el Capítulo 5 ”Transformación Conceptual”. De este modo, cuando se generan los componentes de las actividades, internamente se procesa esta relación y se invoca en módulo encargado de generar los componentes de las herramientas.

Más adelante, en las próximas secciones de este capítulo, veremos que los Awareness pueden ser modelados dentro de las herramientas, a través de la configuración de un atributo llamado “showIn”.

Elemento	**Descripción**	**Transformación Conceptual**	**Aplicación Cliente**	**Aplicación Servidor**
Herramienta (Tool)	Objetos que operan los usuarios.	Contenedor de operaciones y Awareness.	Componentes Angular.	Asociación con las actividades y los procesos.

Tabla 7.5: Tabla de transformación de herramientas.

7.3.6. Transformación Técnica de Rol de Colaboración

Dentro del metamodelo, un rol de colaboración o “Collaboration Role” nos determina a qué elementos, el usuario tiene acceso. Si el usuario puede acceder al elemento, tiene autorización

para verlo y utilizarlo. En el caso que no lo tenga, el sistema no lo hará visible. Existen dos tipos de roles, el que se especifica cuando un usuario se registra, y los roles automáticos que pueden ser asignados durante la ejecución de las actividades.

Lo primero que tiene que hacer un usuario para poder usar el sistema, es registrarse y elegir el rol al que pertenece.

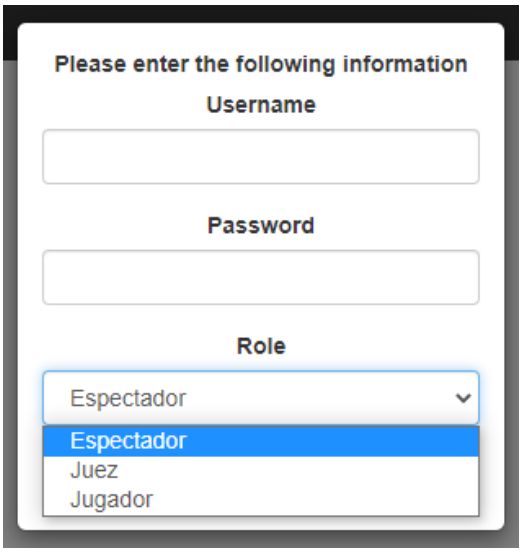

Figura 7.18: Modal de registro.

En la figura 7.18, se muestra un listado de roles que se arma según el modelo. Es responsabilidad del modelador ocultar o no alguno de ellos.

Una vez que el usuario se registra e ingresa al sistema, se le asigna un token que lo acompañará lo que dure su sesión. Este token es llamado JWT ("Json Web Token"), y para este trabajo se utilizó la tecnología PassPort para autenticar y validar la sesión del usuario en ambas aplicaciones. El token se encarga de almacenar el rol que el usuario seleccionó, junto con su nombre de usuario. El cliente y el servidor se comunicaran entre ellos pasando este token para identificar al usuario. Además, nos aporta un extra en seguridad, ya que si el usuario ingresa al sistema mediante las rutas generadas, copiándolas directamente en su navegador, se comprobará este token y si no tiene una sesión válida, será redireccionado a la pantalla principal obligándolo a iniciar sesión correctamente. En caso de tener un token válido podrá acceder a la ruta especificada sin problemas.

El usuario siempre verá el rol al que pertenece en su perfil de usuario, como se puede observar en la siguiente figura 7.19. Pero si participa en una actividad, y existen roles automáticos que

dependen de la ejecución y la configuración, también se verán reflejados a la derecha del nombre de usuario. Si el mismo vuelve a la página principal verá su rol original. Este intercambio de roles se verá reflejado siempre que el usuario navegue entre sus actividades.

Figura 7.19: Cambios de roles.

Los roles dentro del metamodelo están representados por la clase "CollaborationRole" y sus operaciones por la clase "RoleElementOperation". Las actividades mantienen una relación uno a N con las mismas, y a su vez, existe una relación de participación ("ParticipationRelationship") que nos indican que rol participa en determinada actividad. No obstante, los roles se pueden asociar con múltiples elementos colaborativos.

Técnicamente, los roles son elementos que se asocian con todos los componentes que el usuario tiene acceso en la interfaz del sistema web. Por consiguiente, cada elemento sabrá qué roles lo habilitan para su utilización. Como cada elemento dentro de la aplicación del cliente es un componente Angular, a cada uno de ellos se le asignará una configuración con los roles a los que pertenece. Esto se realiza al momento de la creación de los archivos webs de cada elemento, pero para validar que un componente se habilite en base a su rol, se debe comparar con el del usuario que está usando el sistema, que se obtiene a través del token.

Tanto del lado del servidor como del cliente, un usuario tendrá su representación a través de una clase que no solo tienen los atributos como, el nombre de usuario y contraseña, sino que también mantienen el rol que el usuario seleccionó cuando se registró o el que se le otorgó en la asignación automática. A su vez, en ambas partes se crean controladores, servicios y rutas relacionadas al usuario para realizar el manejo del registro, inicio y cierre de sesión e incluso los cambios de rol que pueden ir ocurriendo a lo largo de la participación de los usuarios.

Elemento	Descripción	Transformación Conceptual	Aplicación Cliente	Aplicación Servidor
Rol de Colaboración	Define los elementos que el usuario verá y que podrá utilizar para cumplir su objetivo.	Atributo que acompaña al usuario.	Configuración de componentes Angular. Clase TypeScript del Usuario. Atributo que acompaña al objeto del usuario. Servicios genéricos.	Ruta de API Rest. Esquema de Base de Datos para el Usuario. Atributo que acompaña al esquema del usuario. Controlador y servicios genéricos. Archivos de configuración.

Tabla 7.6: Tabla de transformación de roles de colaboración.

7.3.7. Transformación Técnica de Proceso

Cuando se presentó la transformación técnica de las actividades se comentó, para facilitar la comprensión, que el usuario crea instancia,s pero no se describió como lo realiza. Aquí entran en juego los procesos, que no solo se encargan de establecer la creación de actividades, sino que también definen la navegación del sitio y le dan sentido a todos los elementos mencionados previamente, ya que son el punto de entrada a los mismos.

Al igual que las actividades, el usuario puede crear la cantidad de instancias de procesos que desee, y queda bajo la responsabilidad del desarrollador limitarlo, al igual que el listado de instancias que verá el usuario a medida que vaya generando nuevas

El objetivo de los procesos es permitir al modelador ordenar de forma secuencial el orden de ejecución de las actividades. Toda la navegación del sistema web resultante, dependerá de los procesos, con la excepción de la página de inicio, el registro y el inicio de sesión. Para este trabajo se decidió que las actividades solamente puedan ser ejecutadas dentro de los procesos,

aunque el metamodelo permite modelar actividades con el fin de ejecutarse independientemente, se optó por este camino para darle una mayor interpretación al sistema final.

Para el ejemplo del ajedrez, ya sabemos cómo se ejecutan las actividades y le dimos un comportamiento a la actividad "JugarPartida". Ahora vamos a crear un proceso que nos permita iniciar estas actividades. Su finalidad será simular un campeonato de ajedrez, en donde los participantes deberán jugar varias partidas para lograr ganarlo, lo llamaremos "Campeonato". En él, creamos tres actividades del tipo "JugarPartida" y en ellas también agregaremos usuarios de tipo "Juez" para controlar los movimientos y determinar un ganador. Cuando se está modelando un proceso es posible que existan actividades, del mismo tipo, que se repitan, por lo tanto se le debe asignar un nombre para distinguirlas, como es en este caso, donde vamos a tener tres actividades "JugarPartida".

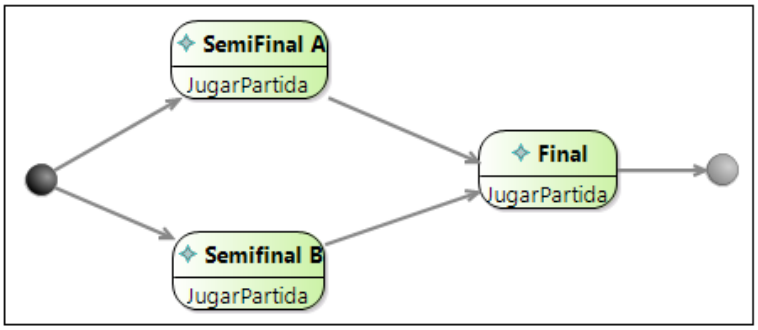

Figura 7.20: Ejemplo de Proceso "Campeonato".

La figura 7.20 muestra el proceso "Campeonato" modelado con el editor de procesos que ofrece el metamodelo. En él podemos observar un círculo gris oscuro que representa el inicio del proceso, indicando que las actividades "SemiFinal A" y "SemiFinal B" son las iniciales, y un círculo gris claro que indica el final del proceso, una vez que termine la ejecución de la actividad llamada "Final". Las actividades iniciales van a comenzar a ejecutarse cuando todos los usuarios del proceso "Campeonato" estén presentes, según la cantidad de participantes configurada para estas actividades. Cuando esta cantidad sea completa, se inician las actividades iniciales, y también, el proceso. La ejecución de las actividades dentro de un proceso es exactamente la misma que la presentada en las secciones 7.3.3 y 7.3.4.

Los componentes que se generan para los procesos, en las tecnologías, tienen una mayor complejidad que los elementos presentados hasta el momento. Por este motivo, se eligió aprovechar las ventajas de las tecnologías para modularizar la funcionalidad de la mejor forma posible.

Para el cliente, Angular se focaliza en dividir la interfaz en pequeñas secciones, y cada una de ellas puede ser un componente según lo que se requiera. Para nuestro caso cada proceso genera dos componentes principales independientes, el primero se encarga de visualizar una entrada en el menú principal, y el segundo gestiona todas las instancias del proceso permitiendo el alta y la consulta de cada una de ellas.

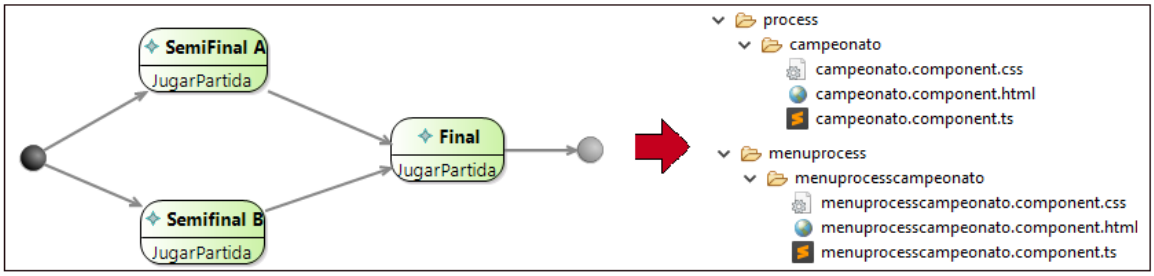

Figura 7.21: Transformación del proceso “Campeonato”.

Al igual que en las actividades, también se forman rutas para acceder a estos componentes y otorgarles navegabilidad. En nuestro ejemplo, se creó la ruta “/campeonato”, para el uso interno del sistema y como punto de acceso para el usuario. Con solo ingresar esta ruta en el navegador y previamente haber iniciado sesión, el usuario podrá consultar directamente las instancias que tiene creadas para este proceso, visualizando su interfaz.

En el servidor, la representación de los procesos está generalizada en su gran mayoría, salvo algunas particularidades. Todas las estructuras comparten sus declaraciones para cada uno de los procesos que se puedan crear con el metamodelo. Se evitó ligar el modelo del proceso con la solución que lo representa, de esta forma si logramos que estas estructuras funcionen para casos específicos de procesos, nos garantizamos que funcionará para los N procesos que se modelen. En primer lugar se creó un esquema genérico en la base de datos para almacenar el contexto de ejecución de los procesos. Luego, las rutas del API Rest que nos ayudan a persistir, consultar y eliminar sus instancias. También, el controlador correspondiente nos posibilita asociar el esquema con las rutas y contiene la lógica necesaria para la sincronización de los usuarios dentro de los procesos.

El contexto de un proceso representa la instancia de ejecución del mismo. Con él podemos conocer qué usuarios están participando, cual es la actividad actual para determinado usuario, cuál es el estado del proceso, entre otros datos. Pero además el proceso debe saber con qué actividades puede iniciar, cuál es la actividad sucesora de la actual, o la actividad final. Todos

estas cuestiones surgieron a lo largo del desarrollo,y para solventarlo se decidió asociar a los contextos una configuración. Si bien en esta sección hablamos de estructuras genéricas, mencionamos que existen particularidades, una de ellas es la configuración, debido a que se generan en base al modelo. A diferencia de la configuración de las actividades, esta configuración no puede ser manipulada por el desarrollador, porque es de uso interno.

Luego de la transformación, si los usuarios tienen los permisos adecuados verán, después de iniciar sesión, una entrada en el menú principal con el nombre del proceso. Desde allí, podrán acceder al listado de instancias de los procesos como así también crear uno nuevo.

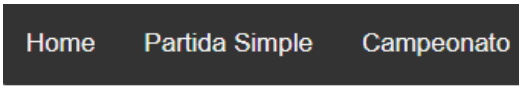

Figura 7.22: Menú para procesos “Partida Simple” y “Campeonato”.

Seleccionar un proceso llevará al usuario a la pantalla de instancias del proceso. Por ejemplo, si el usuario presiona la entrada del menú “Campeonato” verá en el contenedor principal del sistema, la siguiente vista.

Figura 7.23: Instancias de proceso “Campeonato”.

En esta pantalla, el usuario se encontrará con un título principal con el nombre del proceso modelado y un botón que permitirá iniciar una instancia del proceso, el mismo fue agregado al sistema con fines demostrativos. Debajo de este botón “Iniciar Campeonato” podrá observar el listado de instancias en las que participa el usuario, que originalmente comienza vacía.

El listado de la figura 7.23, muestra como título, por cada instancia, el nombre del proceso junto con su identificador ("campeonato 1"), y a su vez, el nombre de la instancia de la actividad ("Semifinal B") en la que participa el usuario. En el detalle de cada instancia se puede visualizar como subtítulo la etiqueta "current activity" ("Actividad Actual") que indica el nombre del tipo de la actividad, en este caso "JugarPartida"; el estado ("state") de la actividad actual; y por último, el listado de participantes ("participants") de la instancia del proceso. Cada fila de este listado tendrá un color dependiendo del estado de la actividad en la que participa el usuario, se pinta de color verde si la actividad esta en estado "running", amarillo si el estado es "pending" y celeste si es "finished". Presionar una instancia llevará a la consulta de la actividad y el usuario podrá operar en ella, dependiendo de su estado, como se presentó en la sección 7.3.3.

En el metamodelo, las clases que representan los proceso son "CollaborativeProcess" y "CollaborativeProcessEdge", esta última se relaciona uno a uno con "RoleElementOperation" y también mantiene una relación uno a N con "CollaborationRole". Todas estas clases se usan en el recorrido para crear los elementos mencionados dentro de las tecnologías.

Elemento	**Descripción**	**Transformación Conceptual**	**Aplicación Cliente**	**Aplicación Servidor**
Proceso	Objetivos que tienen los usuarios para utilizar el sistema.	Secuencia de ejecución de actividades.	Componente Angular. Ruta de navegabilidad. Clase TypeScript.	Ruta de API Rest. Esquema de Base de Datos. Controlador y configuración. Servicios.

Tabla 7.7: Tabla de transformación de procesos.

7.3.8. Transformación Técnica de Awareness

Los Awareness son los elementos encargados de mantener información de las actividades que realizan los usuarios en el sistema. Por ejemplo, los estados de un usuario, es decir, si un determinado usuario se conecta o desconecta, haciendo que el resto de los usuarios vean el

cambio de su estado, inmediatamente. Nos hemos referido a los Awareness como eventos que ocurren en tiempo real, y sirven para visualizar información que se actualiza en el momento que han ocurrido.

En el capítulo donde presentamos las tecnologías, expusimos que una de las tecnologías más importante, que forman el sistema, es el WebSocket. Recordando su definición, esta tecnología nos permite establecer una comunicación bidireccional entre el cliente y el servidor. Una vez que se establece esta comunicación se mantiene abierta lo que dure el uso del sistema por parte del usuario y nos garantiza que cualquier dato que se envíe desde una de sus partes le llegará a su contrapuesta en tiempo real. Por este motivo elegimos esta tecnología para el funcionamiento de los Awareness, ya que conocemos los beneficios que nos aporta.

Un Awareness se puede visualizar en cualquier elemento colaborativo, por ejemplo, en los espacio de trabajos, en las actividades o en las herramientas. En el metamodelo, están representados por la clase "Awareness", que mantiene una relación uno a N, llamada "shownIn", con la clase "CollaborativeElement", padre de todos los elementos colaborativos. Esta relación le permite al modelador indicar en qué elemento se visualizará el Awareness. Por otro lado, a cada Awareness se le puede establecer un tipo, que es modelable mediante la clase "AwarenessKind", a su vez esta clase se asocia con "CollaborativeModel", clase del modelo colaborativo. Esto último significa que el modelo conoce todos los tipos de Awareness modelados.

El universo de los Awareness es muy extenso, y nuestro metamodelo permite modelar un sinnúmero de ellos. Para este trabajo, se decidió optar por crear tres tipos de Awareness. Uno de ellos está pensado de forma genérica para que el desarrollador que use la herramienta pueda extenderlo a su gusto. Para definir estos tipos se realizó una investigación para obtener cuales son los más comunes y útiles dentro de los sistemas colaborativos. Los modeladores tendrán la posibilidad de incluirlos en sus modelos y se encontrarán como resultado, el funcionamiento de los eventos sincrónicos que van ocurriendo a medida que los usuario usan el sistema. Luego, podrán modificar sus apariencias según lo deseen.

En la figura anterior, podemos observar la transformación de un Awareness, llamado "Estado", para los componentes creados en el cliente. El mismo se encuentra asociado a la herramienta "Chat" lo que nos indica que se mostrará dentro de ella. A su vez, dentro del directorio "/awareness" se encuentran dos más, llamados "turno" y "ultima-accion", cada uno de ellos además del nombre y del elemento al que se vinculan, son de uno de los tres tipos desarrollados para

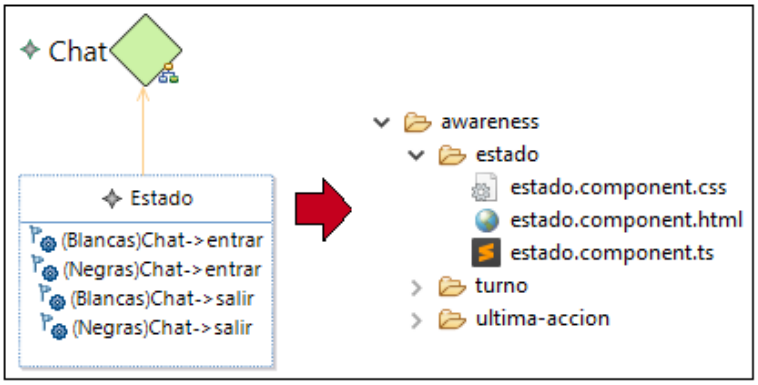

Figura 7.24: Transformación de Awareness.

este trabajo. Dependiendo del tipo que sean, la transformación, la vista y el comportamiento será distinto, como veremos a continuación.

Awareness de Acción

Los tipos de Awareness dependen del alcance del modelo. Existen situaciones en donde es necesario que los participantes conozcan del resto de los usuarios, lo que están realizando o lo que han hecho. Para este caso, se diseñó y creó el Awareness de acción, cuyo objetivo es permitir a los participantes visualizar cual es la última acción o operación realizada, junto con el usuario que la realizó. Este Awareness es una de las posibles variantes que pueden existir para estas situaciones, y la que mejor se adapta a nuestro ejemplo.

Si volvemos a la partida de ajedrez, podría ser muy útil para los usuarios, saber que el jugador contrario o el juez realizó una determinada acción. Por eso, creamos un Awareness de este tipo, y lo llamamos "Última Acción".

En la figura previa, se muestra nuestro Awareness, dentro del editor de la estructura del modelo (parte izquierda de la figura), el mismo al igual que otros elementos colaborativos presentan un atributo "ShownIn" ("Se Muestra en") que nos permite indicar dónde se va a visualizar nuestro Awareness. En este caso, se optó por la herramienta "Tablero", y la vinculación se observa, en el editor, a través de la flujo naranja. Además, los Awareness tienen un atributo llamado "Kind" ("Tipo") donde se indica a qué tipo pertenece, aquí el especificado es "Action" ("Acción"), uno de los tres tipos que se crearon para este trabajo. Por último, también tienen un atributo "Source" ("Fuente") donde se ingresan las operaciones asociadas al Awareness, que

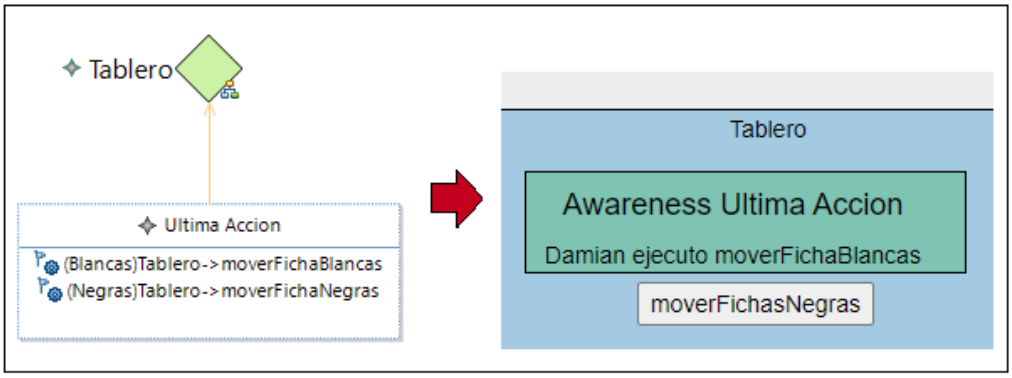

Figura 7.25: Transformación de Awareness “Última Acción”.

a continuación veremos cómo se usan en la funcionalidad resultante.

El resultado del Awareness, luego de haber aplicado la transformación, es el componente de interfaz que podemos observar en un recuadro verde en la parte derecha de la figura 7.25. El comportamiento es muy sencillo e intuitivo, cada vez que los usuarios presionen algunas de las operaciones vinculadas con el Awareness, el resto podrá ver al instante, dentro de la misma caja verde, que tal usuario ejecutó tal operación, como lo muestra el ejemplo de la figura 7.25. Siempre se verá la última operación ejecutada por parte de algún usuario. Las operaciones que se sincronizarán son las modeladas dentro del listado del atributo “Source”, en donde se ingresan instancias de la clase RoleElementOperation, creadas en el momento del modelado, como hemos mencionado en varias ocasiones. Este listado, la herramienta lo convierte en un arreglo de eventos dentro del componente del Awareness. Cuando el mismo se haga visible, los usuarios se suscribirán a ellos y en el caso de que se dispare algún evento, será presentado en la interfaz. Para realizar esta funcionalidad se utiliza la tecnología de WebSockets, para lograr que estos eventos ocurran en tiempo real.

Cuando un usuario ingresa al sistema web, se realiza una conexión bidireccional de WebSocket, esto significa que el cliente y el servidor van a tener un canal abierto por donde comunicarse inmediatamente. Los Awareness aprovechan este beneficio de la tecnología para lograr una navegación en tiempo real. De esta forma, si ocurre un evento, el mismo se propagara por los distintos canales conectados, haciendo que llegue al resto de los usuarios rápidamente.

La tecnología también aporta, en ambas partes, un servidor de WebSockets, en donde se permite gestionarlos, distinguirlos y enviar eventos a cada uno según la ocasión. Existen casos,

en donde no es necesario propagar el eventos a todas las conexiones de WebSockets, no todos los usuarios deben verlos, este tema ya depende según el modelado y los permisos que se le han dado a los participantes. El Awareness de Acción es uno de ellos, ya que los eventos que ocurren sólo deberán procesarlos los que tengan acceso a este elemento y estén participando de la actividad específica. Por lo tanto, para esta situación, cuando ocurren los eventos, el código fuente se encarga de decidir a qué WebSocket debe enviarles el evento. Cada WebSocket se diferencia mediante un identificador único, y se asocia según al usuario según los datos de la sesión.

Por otro lado, cuando el elemento está visible en la interfaz, el cliente se suscribe a los eventos, y cada vez que le llegue uno, vía conexión WebSocket, su código lo procesa según el tipo de Awareness que se esté usando. Esto quiere decir, que el código de cada componente Awareness creado, tendrá un comportamiento y una manera distinta de procesar dependiendo el tipo que sea.

Es importante comentar que este tipo de Awareness fue desarrollado de una manera genérica, esto quiere decir, que el programador que use la herramienta, podrá modificar el contenido que se envía en los eventos, para agregarle lo que necesite. Si bien el Awareness ya viene con un comportamiento específico, este mismo se puede extender de una forma muy sencilla y que se adapte mejor a los requerimientos del que emplea la herramienta. No importa cual sea la estructura que se configure para enviar en los eventos de WebSocket, el sistema garantiza que siempre les llegará a los participantes correspondientes.

Todas las interfaces visuales de los Awareness que se presentan en esta sección, son estéticamente muy simples, ya que son los elementos que más se van a configurar y modificar en cuanto a su diseño, porque son visibles y utilizados durante la navegación del sistema.

Awareness Temporizador

Este tipo de Awareness permite al modelador jugar con el tiempo y las transiciones de estados que pueden surgir durante la ejecución de una actividad. El objetivo es poder realizar automáticamente estas transiciones, es decir, ejecutar alguna operación, luego de que haya pasado un determinado tiempo configurado. Por lo general, este tipo de Awareness es utilizado mayoritariamente en juegos online, en donde los jugadores deben realizar alguna acción antes

que se termine un temporizador.

En este caso, durante el modelado, se podrán agregar, nuevamente dentro del editor, Awareness del tipo “Timer” (“Temporizador”). Para nuestro ejemplo principal, podemos crear uno, llamado “Turno”, que controle el tiempo de respuesta de los jugadores en cada movimiento de ficha. De esta forma, si los jugadores no mueven una ficha, la acción se ejecutará una vez que el contador llegue a cero.

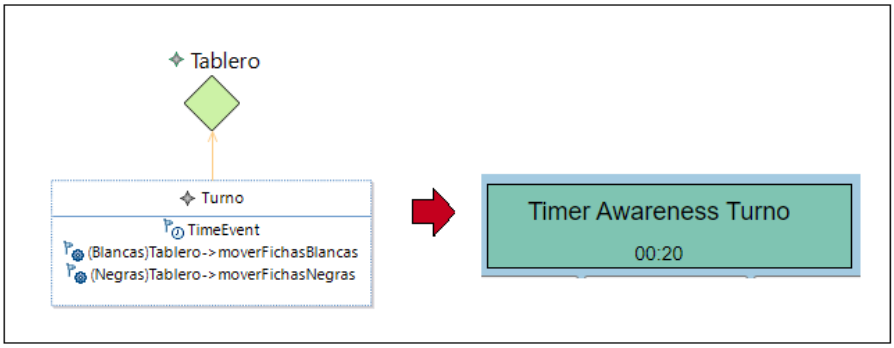

Figura 7.26: Transformación de Awareness “Turno”.

En la figura 7.26, se presenta el modelado (parte izquierda) y el componente de la interfaz (parte derecha) para este tipo de Awareness. Si a un usuario del rol “Blancas” o “Negras”, le toca mover una ficha, deberá realizarlo previo a los veinte segundos que se muestran en el rectángulo verde. En el caso contrario, el sistema ejecutará automáticamente la operación. Esto permite mantener un mayor control, en este caso, de la partida, ya que permite que la transición de estados de la actividad no se quede estancada por un usuario particular. Cuando le toque nuevamente al que se le terminó el turno, el contador comenzará una vez más. Aquí presentamos un ejemplo aplicado a nuestro ajedrez, pero este Awareness puede ser útil para cualquier estado que requiera un contador de tiempo. Inclusive se puede extender su lógica para que también, además de mostrar el temporizador, se pueda ver algunos datos adicionales.

Como todos los Awareness que veremos en esta sección, este tipo se representa también con un rectángulo verde, logrando una distinción con el resto de los elementos del sistema.

Awareness de Presencia

Este tipo de Awareness es el más popular dentro de los sistemas colaborativos. Su objetivo es informar la presencia del usuario al resto de los participantes. Esto significa, que los

participantes conocerán si un determinado usuario está en línea, es decir, si está conectado en ese momento o no. Esta funcionalidad es muy útil en sistemas colaborativos empresariales enfocados en reuniones virtuales, donde es necesario saber quién está presente en determinados momentos; o por ejemplo en el ámbito de los videojuegos, en donde los jugadores pueden darse cuenta si el rival o los que forman parte del equipo esta conectados, o si alguno, por cualquier motivo, no esta presente. Es un Awareness que cualquier sistema colaborativo debe tener, porque logra darle a los usuarios la sensación de que están trabajando de forma cooperativa, como si estuvieran presentes físicamente.

En nuestra herramienta, modelar un Awareness de este tipo es tan simple como usar el editor del metamodelo, crear un Awareness, asignarle un nombre y el tipo "Presence" ("Presencia"). Luego, como el resto de los tipos de Awareness, se debe indicar mediante el atributo "ShownIn" donde se va a mostrar.

El resultado de la transformación, es un nuevo componente en la interfaz, que se encarga de gestionar los eventos de los participantes que tienen acceso a la ejecución de la actividad.

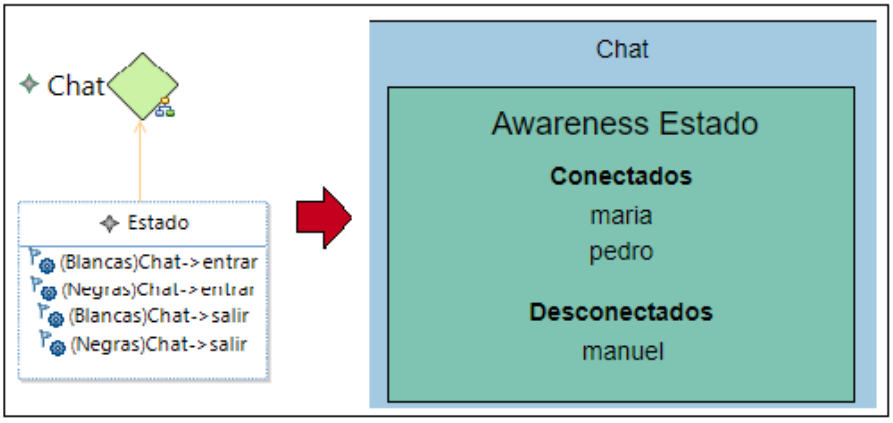

Figura 7.27: Transformación de Awareness "Estado".

En la figura anterior, se creó un Awareness de tipo "Presence", llamada "Estado", que se muestra dentro de la herramienta "Chat". En la sección derecha, se muestra el componente creado luego de aplicar la transformación, en ella, se pueden observar los usuarios en el estado "Conectados" y en "Desconectados". Un usuario puede cambiar de estado, dependiendo de las operaciones que ejecute, y que previamente se asignaron en el modelado del Awareness.

Por otro lado, no solo siempre se necesita saber si los usuarios están en estado "Conectados" o "Desconectados", en determinadas ocasiones, estos estados pueden cambiar. Sabiendo esto,

y como siempre tenemos el foco en que debe funcionar para cualquier modelo, este Awareness permite configurar, al modelador, los estados de los usuarios dentro del mismo. Por defecto, viene configurado con los estados “Conectados” y “Desconectados”, pero el modelador podrá cambiarlo cuando lo desee, en una configuración que se encuentra dentro del componente del cliente, para este Awareness. Como por ejemplo, en un modelo de una clase virtual, donde la actividad es “RendirExamen”, se podría poner como estado “RealizandoExamen” y que nos muestre todos los alumnos que están rindiendo. Cuando un alumno presione la operación “Finalizar”, dejará de aparecer en el listado de los usuario del estado “RealizandoExamen”, y pasará a “Finalizado”. De esta forma, el docente podrá saber en tiempo real cuántos alumnos están rindiendo y cuantos finalizaron.

Este Awareness tiene lo necesario para mostrar la presencialidad de los usuarios en el sistema, pero además, gracias a los eventos y los datos que maneja, puede ser fácilmente extensible según se requiera. Incluso, se podrían crear múltiples de este tipo de Awareness para separar de diferentes formas o clasificación a los usuarios dentro de las actividades.

Como el resto de los tipos de Awareness presentados, aquí también se distribuyen los eventos internos mediante el WebSocket, aprovechando que se mantiene abierto durante la sesión del usuario.

Elemento	**Descripción**	**Transformación Conceptual**	**Aplicación Cliente**	**Aplicación Servidor**
Awareness	Elementos que se encargan de visualizar información de los usuarios.	Eventos automáticos relacionados a las acciones de los usuarios.	Componente Angular. Eventos de datos. Servicio genérico de WebSocket.	Servicio genérico de WebSocket. Esquema de Base de Datos. Controlador de los Esquemas de WebSocket.

Tabla 7.8: Tabla de transformación de Awareness.

Capítulo 8

Beneficios de la Herramienta y Conclusión

8.1. Introducción

En este capítulo, listamos y explicamos brevemente cada uno de los beneficios que se obtienen al usar la herramienta de transformación de sistema colaborativos presentada. Estas ventajas evitan que el desarrollador o modelador deban realizar funcionalidades que son comúnmente encontradas dentro de este tipo de sistemas. Además, junto con el metamodelo de sistemas colaborativos se logra modelar y crear múltiples aplicaciones web colaborativas a muy bajo costo.

Por último, se podrá leer una conclusión general de todo lo investigado y desarrollado para abordar este trabajo.

8.2. Beneficios de la Herramienta

8.2.1. Administración de Usuario

El sistema web resultante dispone de funcionalidades relacionadas con el usuario. Por lo tanto, el modelador no debe preocuparse por el manejo de la sesión ni por la autenticación de

los usuarios. El sistema ya tiene incorporado el registro del usuario y el inicio/cierre de sesión del mismo. Además, aporta un nivel de seguridad en las rutas generadas, ya que comprueba, mediante el uso del token, la sesión del usuario en cada acceso.

8.2.2. Manejo de Roles

Los roles que se crean en el metamodelo son interpretados por el sistema web y permiten al modelador gestionar diferentes perfiles, determinando qué elementos serán visibles y a cuales tendrán acceso los distintos usuarios. También brinda la posibilidad de configurar la cantidad de usuarios que participan en un rol y cuáles son más concluyentes. Toda esta funcionalidad se puede aprovechar sin necesidad de agregar código propio.

8.2.3. Estructura de Aplicaciones y Entorno de Desarrollo

Luego de la transformación, cada uno de los elementos del modelo pueden generar componentes en las distintas aplicaciones, es decir, en la aplicación del cliente y en la que funciona como servidor. La herramienta otorga una estructura de directorios, en cada aplicación, que permite organizar y reconocer de forma ágil cada uno de los componentes generados para asociarlos inmediatamente con su correspondiente elemento del modelo. Por otro lado, la interfaz es presentada de una manera intuitiva y sencilla para poder distinguir cada uno de ellos, a través de colores y subtítulos.

Además, las aplicaciones, junto con la base de datos, se pueden iniciar mediante comandos. Ambas, cliente y servidor, quedan pendientes de los cambios que se realicen en la transformación, es decir, que las modificaciones del modelo se toman instantáneamente. Por lo tanto, si por cada cambio que realicemos ejecutamos la transformación nuevamente, los mismos se verán reflejados automáticamente en el browser. Esto logra que iniciar el entorno de desarrollo y trabajar en él, facilite la implementación del sistema resultante.

8.2.4. Armado de Procesos, Actividades y Operaciones

La creación de procesos y actividades colaborativas, junto con sus protocolos de estados, permiten definir gran parte de la navegación del sistema resultante. Estos elementos colaborativos definen la estructura del menú principal, es decir, los puntos de entradas a la ejecución. De esta forma, se da total libertad para definir el flujo de navegación del sistema, mediante el modelo.

Por otro lado, las operaciones dan la posibilidad de controlar la ejecución. Por medio de ellas, se pueden modelar todos los eventos o acciones que los usuarios pueden realizar cuando se encuentran usando el sistema.

8.2.5. Sincronización de Usuarios

El sistema resultante se encarga de gestionar la ejecución de los procesos y las actividades colaborativas. Esto se realiza a través de la creación de instancias de cada elemento. Además, el sistema administra los diferentes estados internos por el cual pueden pasar estos elementos colaborativos para sincronizar los usuarios según vayan participando y accionando las diferentes operaciones modeladas.

8.2.6. Cooperación de Usuarios

Otro beneficio que se obtiene es la cooperación de usuarios. Esto significa que la combinación de los procesos, las actividades, las operaciones, más la sincronización de los usuarios, logra que los usuarios puedan cooperar y trabajar de manera conjunta en sus actividades. Logrando cumplir con el fin de un sistema colaborativo.

8.2.7. Awareness

Por último, uno de los beneficios más importantes de la herramienta es poder crear Awareness de diferentes tipos, coordinando los eventos en tiempo real a medida que ocurren y darle la posibilidad al modelador de poder extender su funcionalidad, ya que fueron creados de forma genérica. Y de esta forma, cumplir con el objetivo de crear un sistema colaborativo completo según el modelo.

8.3. Conclusión

Para concluir, presentamos nuestra opinión y la experiencia que se obtuvo al elaborar este trabajo, durante las etapas del proceso de desarrollo, es decir, investigación, estimación, implementación y prueba.

El objetivo siempre estuvo centrado en crear una herramienta de transformación de sistemas colaborativos en base a un metamodelo. Pero la etapa de investigación nos llevó a introducirnos en este tipo de sistemas y entrar en profundidad en la metodología MDD, para encontrar las mejores prácticas y poder demostrar los beneficios de cada uno de ellos. En base a esto, y una vez definida la mejor manera de enfrentar el desarrollo, se comenzaron a investigar las tecnologías de la actualidad para definir cuáles eran las mejores opciones. Analizando las ventajas que nos podían aportar, siempre teniendo presente nuestra estrategia de desarrollo. La etapa de investigación nos dejó una buena base para comenzar la implementación.

La etapa de estimación comenzó luego de obtener los objetivos y las tecnologías bien establecidas. Aquí nos concentramos en definir el alcance, sin salirnos de lo que queríamos demostrar. A su vez, buscamos fijar un ejemplo sencillo y habitual, dentro de los sistemas colaborativos, que nos permita ser explicativos durante la redacción.

La etapa de implementación y prueba fue bastante extensa. Como es de esperar en un desarrollo, surgieron imprevistos y cuestiones que se debían analizar, y luego tomar una decisión que cumpla adecuadamente con los requisitos definidos. En cuanto a las tecnologías, no se presentaron muchos inconvenientes, ya que en algunas, se tenía un conocimiento previo, solo se debían acoplar juntas y comprobar que funcionen como se esperaba. Esta etapa nos ayudó a poner en práctica cada una de las ventajas que nos aporta la metodología, conocidas durante la investigación.

En términos generales, con respecto a todo el trabajo realizado, se pudo desarrollar una herramienta que permita la transformación de un modelo colaborativo, creado por el metamodelo presentado, y aplicar los conceptos y prácticas de la metodología MDD. Como resultado, no solo se obtienen los beneficios de la herramienta mencionados en la sección anterior, sino también todas las ventajas que aporta la metodología, y un sistema web colaborativo generado en base al modelo.

Bibliografía

[1] L. M. Bibbo, R. Giandini, and C. Pons, "DSL for collaborative systems with awareness," in *2017 43rd Latin American Computer Conference, CLEI 2017*, vol. 2017-Janua, pp. 1–9, 2017.

[2] L. M. Bibbo, R. Giandini, and C. Pons, "DSL for Collaborative Systems with Awareness," *SLISW - Simposio Latinoamericano de Ingeniería de Software; XLIII CLEI - Conferencia Latinoamericana de Informática*, 2017.

[3] H. D. Xihui Zhang, Tao Hu and X. Li, "Software Development Methodologies, Trends, and Implications," 2010.

[4] R. S. Pressman, "Ingenieria del Software. Un Enfoque Practico," tech. rep., 2010.

[5] C. Pons, "El proceso de desarrollo de software basado en modelos," 2010.

[6] M. Brambilla, J. Cabot, and M. Wimmer, "Model-Driven Software Engineering in Practice," *Synthesis Lectures on Software Engineering*, vol. 1, pp. 1–182, 9 2012.

[7] G. Gartner, "Gartner, Empresa consultora y de investigación de las tecnologías. Stamford, Connecticut, Estados Unidos," 1979.

[8] L. M. Bibbo, "Modelado de Sistemas Colaborativos," *Tesis de Magister en Ingeniería de Software; Universidad Nacional de La Plata*, pp. 1–143, 10 2009.

[9] C. Larman, *UML y Patrones Una introducción al Análisis y Diseño Orientado a Objetos y al Proceso Unificado*, vol. Segunda. Prentice Hall, 2002.

[10] S. Tayib, "Client-Server Model. University Utara Malaysia, Kedah, Malaysia,"

[11] "Protocolo de Transferencia de Hipertexto (Hypertext Transfer Protocol), https://www.w3.org/Protocols/."

[12] “Lenguaje de Marcado Extensible (Extensible Markup Language), https://www.w3.org/XML/.”

[13] “Notación de Objeto de JavaScript (JavaScript Object Notation, JSON), https://www.w3.org/TR/json-ld11/.”

[14] “Protocolo de Acceso a Objeto Simple (Simple Object Access Protocol, SOAP), https://www.w3.org/TR/soap/.”

[15] F. L. Marc Huffmeyer, Florian Haupt and U. Schreier, “Authorization-aware HATEOAS. Universitat Stuttgart, Stuttgart, Germany,”

[16] “Lenguaje de Marcas de Hipertexto (Hypertext Markup Language, HTML), https://www.w3.org/TR/html52/.”

[17] “JavaScript, https://www.w3.org/standards/webdesign/script.”

[18] “TypeScript, https://www.typescriptlang.org/.”

[19] “JavaScript Prototypes, https://dev.w3.org/html5/spec-LC/infrastructure.html.”

[20] D. Riehle, “Framework Design, A Role Modeling Approach. Swiss Federal Institue of TechnologY Zurich ,” 2000.

[21] “NodeJs, https://nodejs.org/es.”

[22] “Manejador de Paquetes de Node (Node Package Manager, NPM), https://www.npmjs.com/.”

[23] F. David, “Aplicaciones de una Sola Página (Single-Page Application, SPA) Flanagan, David, ”JavaScript - The Definitive Guide”, 5th ed., O’Reilly. ,” 2006.

[24] S. Pope, “A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk80 System. University of California, Santa Barbara ,” 1988.

[25] D. E. Bakken, “Middleware. Washington State University,” 2001.

[26] “Identificador de recursos uniforme (Uniform Resource Identifier, URI), https://www.w3.org/Addressing/URL/uri-spec.html.”

[27] “JSON Binario (Binary JSON), https://www.mongodb.com/blog/post/bson.”

[28] V. G. Cerf and R. E. Kahn, "A Protocol for Packet Network Intercommunication," 1974.

[29] Y.-L. H. Fu-Hau Hsu and K. Chang, "TRAP: A Three-Way Handshake Server for TCP Connection Establishment. Department of Computer Science and Information Engineering, National Central University, Taoyuan," 2016.

[30] Acceleo — Home, "https://www.eclipse.org/acceleo/."

[31] "Eclipse Modeling - MDT - Home, http://www.eclipse.org/modeling/mdt/."

[32] "Object Management Group (OMG), https://www.omg.org/."

[33] "Entorno de Desarrollo Integrado (Integrated Development Environment, IDE), https://www.sciencedirect.com/topics/computer-science/integrated-development-environment."

[34] "Herramientas de Desarrollo de Java (Java Development Tools), https://www.eclipse.org/jdt/."

[35] L. Bibbó, "Diseño de un Laboratorio Remoto Colaborativo para Robótica Educativa usando el Lenguaje de Modelado CSSL. Sistemas Colaborativos, Facultad de Informática, Universidad Nacional de La Plata.."

Anexo A

Estructura de la Herramienta

A.1. Introducción

El desarrollo de este trabajo implicó la creación de tres aplicaciones, en donde cada una de ellas abarca una tecnología particular. La aplicación de transformación es el proyecto principal, y es el encargado de recorrer el modelo colaborativo e ir transformando cada elemento del mismo. Su tecnología es Acceleo, y la estructura de directorios primordial está dada por el formato propio de esta tecnología, con algunas variantes adaptadas para este trabajo, que se detallan en la próxima sección. Cómo su objetivo es aplicar la transformación, se debe especificar dónde se van a almacenar estos archivos generados a partir del modelo. En nuestro caso los archivos producidos contienen código web, que se divide en otras dos tecnologías, cliente y servidor. El directorio, donde se almacenan los archivos generados, es parte del proyecto de Acceleo, por lo tanto, engloba a las otras dos. Cada transformación que se realice sobreescribirá los archivos de estas aplicaciones.

Para sintetizar la idea, tenemos una aplicación principal, que se encarga de la transformación y que contiene a su vez dos aplicaciones, cliente y servidor, que serán el destino de los archivos generados después de la transformación. En las siguientes secciones se expondrá la estructura de cada una.

A.2. Estructura de la Aplicación de Transformación

La tecnología que se ocupa de realizar la transformación de nuestro modelo es Acceleo, que es un lenguaje de transformación utilizado para convertir un modelo a texto, como se mencionó en el Capítulo de Tecnologías. Provee un lenguaje de programación basado en etiquetas, cumpliendo con las estructuras de condiciones básicas y la posibilidad de declarar variables mientras se recorre el modelo. No se entrará en detalle sobre este tema pero si se comentará los componentes de Acceleo. Sí se quiere obtener más información sobre Acceleo, en la biografía de este trabajo se deja el enlace a su documentación.

Previamente mencionamos que nuestro proyecto Acceleo es la aplicación principal, y que sigue la estructura de directorios otorgada por sus propios creadores. En la siguiente imagen se muestra un ejemplo de esta estructura.

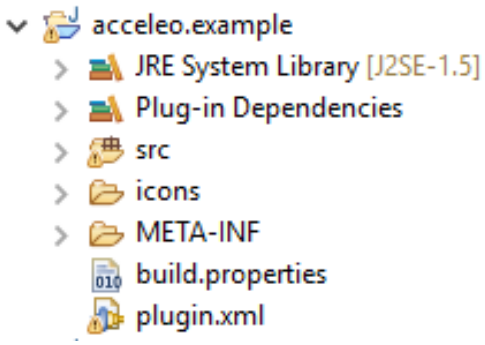

Figura A.1: Estructura de Directorios de Proyecto Acceleo.

Procedemos a explicar brevemente sus directorios/archivos:

- **JRE System Library:** Contiene todas las librerías de Java.
- **Plug-in Dependencies:** Plugins propios de Acceleo. Para este trabajo se agregó como dependencia en esta carpeta, el proyecto de clases de nuestro metamodelo.
- **Src:** Aquí se almacenan todos nuestros módulos de Acceleo. En los próximos párrafos explicaremos qué son y para qué sirven.
- **Directorios Icons y META-INF:** Contienen el icono por defecto de Acceleo y el archivo MANIFEST.MF, utilizado para la configuración del proyecto, respectivamente.
- **Build.properties:** Archivo que permite configurar las propiedades de la transformación.

- **Plugin.xml:** Contiene configuraciones de los Plugins usados por Acceleo.

Nuestro proyecto Acceleo se llama “org.lifia.collaborative-tool” y cumple con la estructura por defecto, pero se crearon tres nuevos directorios, para facilitar el uso de la herramienta y mantener el orden de la misma.

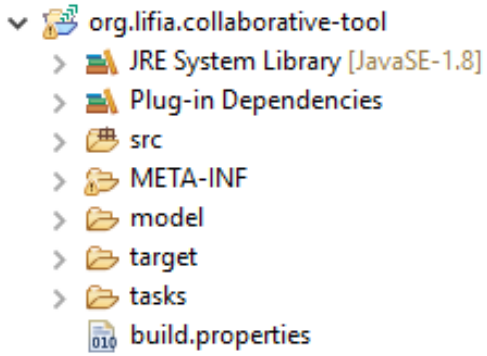

Figura A.2: Estructura de Directorio de Aplicación “org.lifia.collaborative-tool”.

La imagen anterior muestra los nuevos directorios, llamados “model”, “target” y “task”, creados para este trabajo.

- **Model:** Directorio que mantendrá todos los modelos que el desarrollador puede crear con el metamodelo. El objetivo de este directorio es mantener todos los modelos en un solo lugar. El proyecto está configurado para obtener el modelo desde esta carpeta. En la siguiente imagen se pueden ver distintos modelos colaborativos creados con el metamodelo.

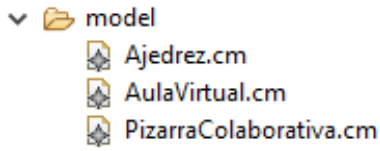

Figura A.3: Ejemplo de Modelos en Directorio Model.

- **Target:** La finalidad de este directorio es ser el destino de los archivos que genera la transformación. Se ha mencionado que este proyecto engloba a las otras aplicaciones, cliente y servidor. Las mismas se almacenan en este directorio, como se visualiza en la figura A.4. Siempre dijimos que nuestra transformación crea archivos web. Estos archivos pueden ser con extensión TypeScript, HTML, CSS o incluso JSON. Todos estos formatos

hacen a nuestro sistema web colaborativo y se distribuyen tanto del lado del cliente como también del lado del servidor. Cuando hablamos de archivo web, dentro de este trabajo, nos referimos a un archivo que sea de alguna de las extensiones nombradas. En la imagen se ven dos nuevos directorios, "collaborative-tool" es la aplicación del cliente, y es donde se almacenan los archivos relacionados a él. Por otro lado, el directorio llamado "server" contendrá la aplicación del servidor, en donde también se generan archivos web.

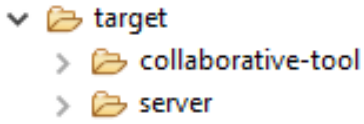

Figura A.4: Directorio Target.

- **Tasks:** Directorio que mantiene archivos XML de configuración pertenecientes al proceso de transformación. Estos dos archivos le indican a Acceleo de qué directorio debe tomar el modelo y cuál es la carpeta destino de sus archivos.

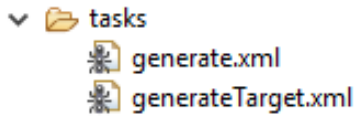

Figura A.5: Directorio Tasks.

Un módulo Acceleo, es un archivo con extensión .mtl que contiene templates (plantillas con nuestro código) y/o queries (consultas, para extraer información del modelo manipulado). Estos módulos nos permitirán ir recorriendo el modelo y a su vez generar los archivos necesarios para que nuestro sistema web funcione como lo deseamos. Los archivos .mtl se guardan dentro del directorio "src", no obstante no tiene por defecto una organización, así que se necesitó definir una estructura de paquetes para mantener una disposición adecuada de estos archivos, siguiendo siempre las buenas prácticas recomendadas por los especialistas de MDD y de Acceleo.

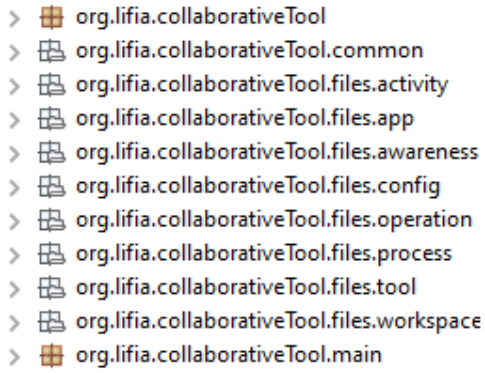

Figura A.6: Estructura de paquetes de los módulos de nuestro proyecto Acceleo dentro del directorio "src".

- **main:** Contiene todos los módulos con templates principales, y es donde se mantiene el módulo (generate.mtl, módulo principal) y la clase principal (Generate.java) que lanza la generación del código.

- **common:** Contiene todos los módulos o queries útiles para el resto de los módulos.

- **files:** Contiene todos los módulos que generan archivos. En nuestro caso, se generarán archivos con extensión .ts, .html y .css. Además, dentro del mismo, se encuentran directorios que distinguen a cada elemento del metamodelo.

La sucesión de pasos que realiza Acceleo para aplicar la transformación es muy simple. Lee el modelo buscándolo en el directorio "model". El modelo leído se usa de entrada para el módulo principal "generate.mtl", a su vez este módulo se encarga de desencadenar las llamadas a los demás módulos, que se encuentran dentro del directorio "src". Los módulos que generan archivos web los almacenan en su correspondiente aplicación dentro del directorio "target".

A.3. Estructura de la Aplicación del Cliente

Ahora procedemos a comentar cómo será la estructura de paquetes que formará la aplicación del cliente. En el apartado anterior, se comentó que dentro de un directorio llamado "target" del proyecto principal se guardarán los archivos web generados por la herramienta. Dentro de este directorio, encontramos la aplicación del cliente, llamada "collaborative-tool". Todos los

archivos que se generen y estén relacionados con la interfaz del usuario serán parte de esta aplicación.

El framework seleccionado para el cliente es Angular. La aplicación del cliente cumple con la estructura estándar de la tecnología. No entraremos en detalle sobre esta estructura, solo comentaremos brevemente cual es el fin de cada directorio. En la bibliografía se podrá encontrar un enlace a su documentación, si se desea leer más del tema.

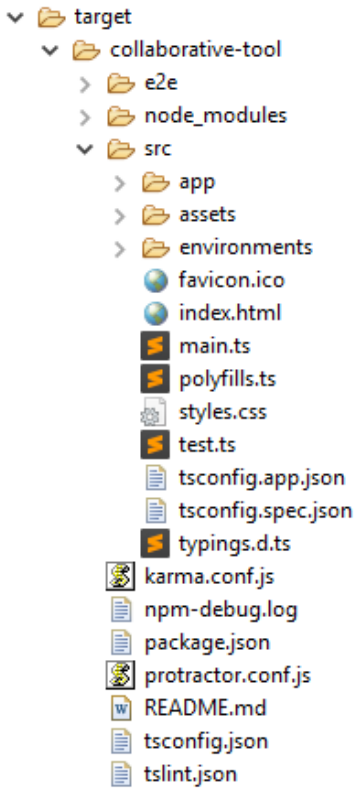

Figura A.7: Estructura de la aplicación del cliente.

- **node_modules:** Directorio que almacena todas las dependencias del framework y las de terceros.
- **e2e:** Directorio asociados a las pruebas unitarias.
- **src:** Es la carpeta más interesante. En ella se almacenan todos los archivos relacionados con el código de la aplicación. Se compone a su vez de 3 subdirectorios:
  - **app:** Aquí se guardan los archivos TypeScript, las vistas HTML y los archivos CSS de la aplicación. Esta carpeta será destinada a los archivos generados con la herramienta. Más adelante explicaremos más en detalle sus subdirectorios.

- **assets:** Almacena los archivos estáticos usados en la aplicación, por ejemplo las imágenes.
- **environments:** Contiene archivos de configuración de la aplicación, como por ejemplo la url al servidor, o el estado de la aplicación, si está en producción o en desarrollo.

- El resto de los archivos que se visualizan en la figura A.7 son todos archivos de configuración de Angular y TypeScript. No está al alcance de este trabajo explicarlos.

Dentro del directorio "src/app" se encuentran todos los archivos generados y asociados al modelo. Angular no aporta una estructura para el directorio "app", por lo general queda en manos del desarrollador.

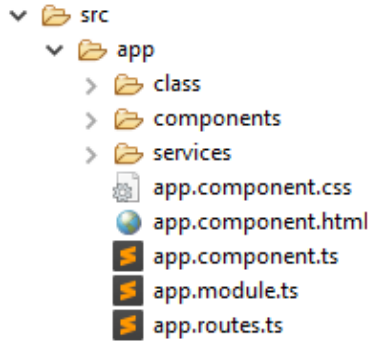

Figura A.8: Estructura directorio "app" de la aplicación del cliente.

La imagen A.8 muestra la estructura de directorio que se determinó para este trabajo, que está compuesta por varios archivos y subdirectorios.

- Los archivos "app.component.ts", "app.componente.html" y "app.component.css" forman parte del componente principal de la aplicación. Este será el primer componente que se carga, y es el que carga los hijos. Por lo general es el que tiene la vista principal de la página.

- **app.module.ts:** Este archivo es utilizado para declarar todos los componentes de la aplicación. Le indica a Angular qué componentes forman parte de la aplicación web. Si no está declarado aquí, no se tendrá en cuenta. Existen componentes que se acceden vía rutas. Todas las rutas se definen en el archivo "app.routes.ts".

- **class:** Este directorio mantiene todos los archivos de clases. Con ellas se crean las instancias de los objetos que se manejan y se transmiten entre los componentes. Por ejemplo, el

objeto “user.ts” que se usa en el servicio de autenticación, o “process-context.ts” usado para sincronizar los procesos junto con el servidor.

- **components:** Contendrá todos los componentes de Angular que genera nuestra herramienta de transformación. Recordemos que un componente Angular es un conjunto de archivos (HTML, CSS, TS) que representan un determinado elemento gráfico de la vista con su correspondiente comportamiento.

- **services:** Este último directorio, conserva todos los servicios del cliente. Hay servicios que se comunican con el servidor para consultar o persistir datos, a través del protocolo HTTP, y otros que solo se usan en el cliente, y se comparten entre los distintos componentes.

Como no sabemos el tamaño de los modelos, existe la posibilidad de que se cree una cantidad importante de archivos. Por esta razón, se decidió esta organización. Además, le facilitamos las tareas de desarrollo y mantenimiento al que use la herramienta.

A.4. Estructura de la Aplicación del Servidor

La última aplicación es el proyecto del servidor. Al igual que la aplicación del cliente, también se encuentra en ”target”, dentro de nuestro proyecto Acceleo, y su directorio es llamado “server”. La aplicación del servidor está compuesto por la estructura que brinda el framework Express.

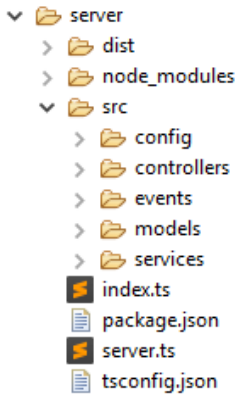

Figura A.9: Estructura de la aplicación del servidor.

En el primer nivel encontramos una estructura muy similar al proyecto Angular.

- **dist:** Directorio que almacena nuestro build productivo una vez que se aplica la transformación. Nuestro servidor va a estar pendiente de los cambios de sus archivos, y se auto-compilará el proyecto generando el build dentro de este directorio.
- **node_modules:** Al igual que Angular, en este directorio se mantienen todas las librerías del frameworks y sus dependencias.
- **src:** Directorio con nuestro código fuente. Todo lo que genere la transformación relacionado al modelo se guarda en este directorio. En esta aplicación, se utilizan solo archivos TypeScript y JSON.
- Archivos "index.ts", "server.ts", "package.ts" y "tsconfig.json" se encargan de la configuración del servidor, de cómo se realiza la transpilación de TypeScript a JavaScript y de las configuraciones propias del framework Express y de la base de datos Mongo.

Como se puede notar en la figura A.9 la carpeta "src" también se compone de directorios. Con el mismo objetivo, mantener una organización lo más limpia posible, para orientar al que usa la herramienta. Y de esta forma, permitirle agregar sus propias estructuras, según lo desee, y facilitarle el mantenimiento.

- **config:** Contiene archivos relacionados a la configuración del modelo. Por defecto, incluye la configuración de la ejecución de los procesos.
- **controllers:** Contiene los controladores encargados de atender y procesar las solicitudes de los endpoints definidos en base al modelo.
- **events:** Contiene archivos relacionados a los Awareness. Este directorio almacena todo lo asociado a ellos.
- **models:** Contiene los archivos que representan los esquemas utilizados para la base de datos.
- **services:** Contiene los servicios que utilizan los controladores para leer la configuración de la ejecución.

Anexo B

Ejemplo de Robótica

B.1. Introducción

En este apartado, presentamos un ejemplo de un modelo utilizado para simular una clase virtual de robótica. En él, participan docentes y alumnos, y su objetivo es permitir al docente gestionar la clase, y a sus alumnos, poder programar, testear y corregir sus códigos de manera online. El mismo fue presentado en el trabajo "Diseño de un Laboratorio Remoto Colaborativo para Robótica Educativa usando el Lenguaje de Modelado CSSL" [35] para la materia "Sistemas Colaborativos" de la Facultad de Informática. No se entra en detalle en este modelo, pero si se mostrará el resultado final de la transformación, para obtener una visión general. Si se quiere obtener una lectura completa, en la bibliografía se deja un enlace con el trabajo. También se logra demostrar que la herramienta cumple y funciona para modelos más complejos.

B.2. Transformación del Ejemplo

En esta sección presentamos las figuras de la transformación para este modelo, se recomienda tener una previa lectura del trabajo para comprender la problemática del mismo.

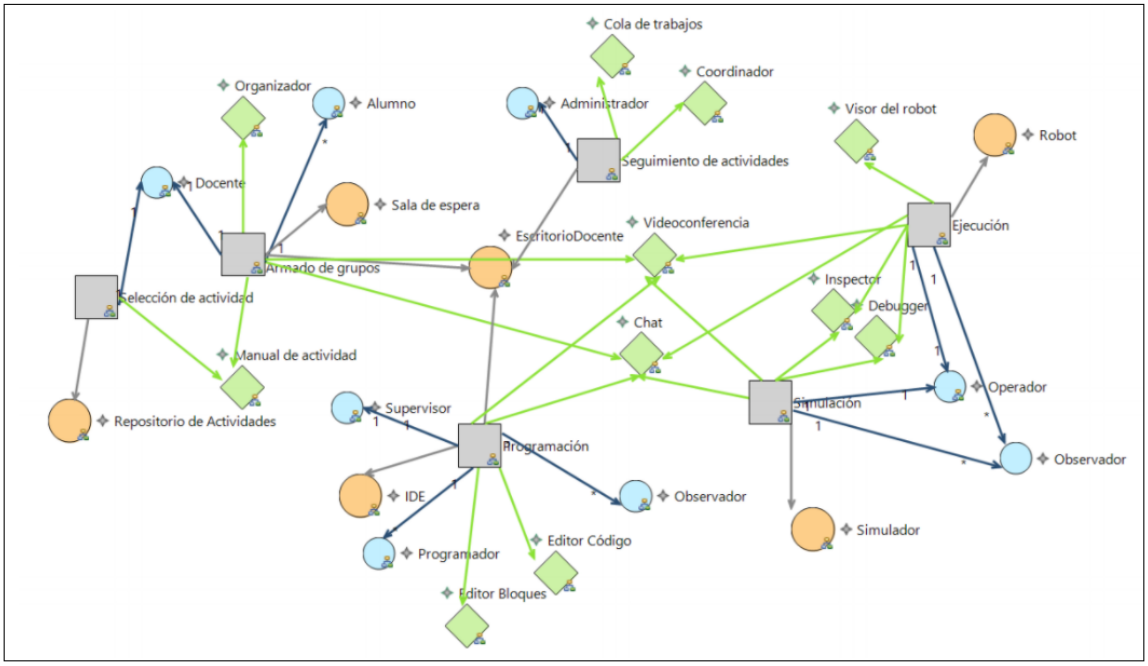

Figura B.1: Modelo de clase de robótica.

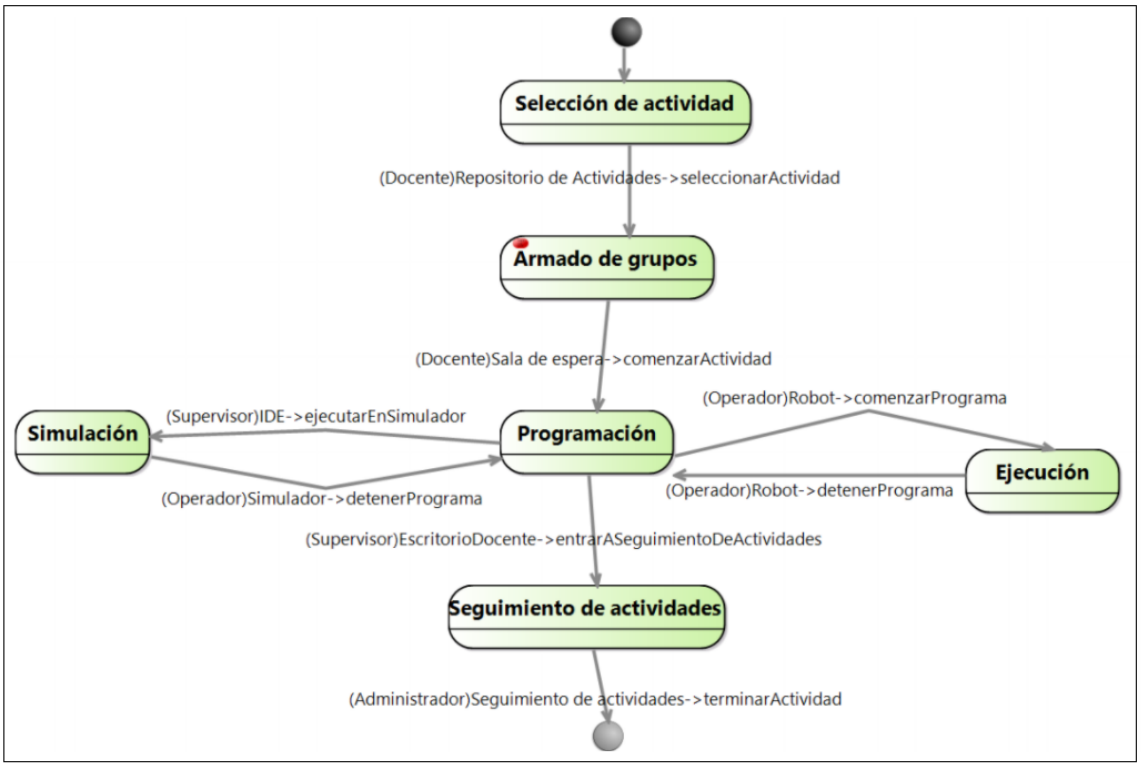

Figura B.2: Proceso de clase de robótica.

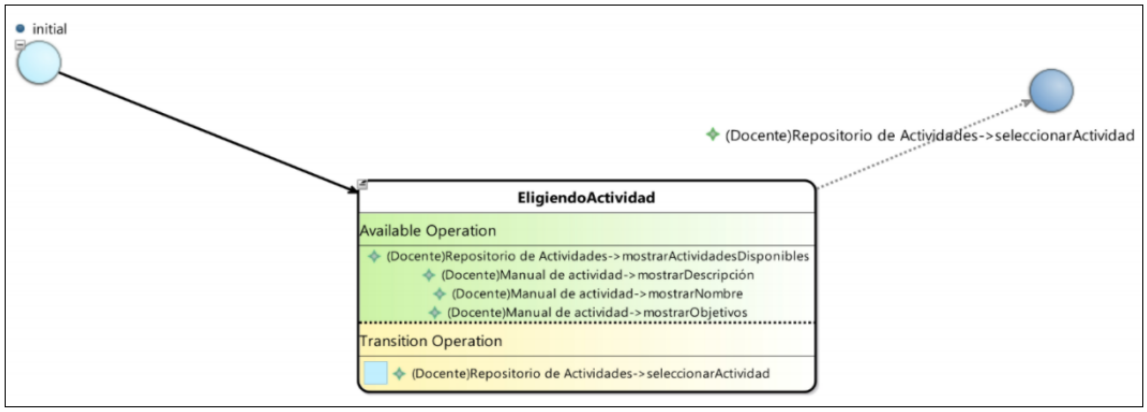

Figura B.3: Actividad “Selección de Actividad”.

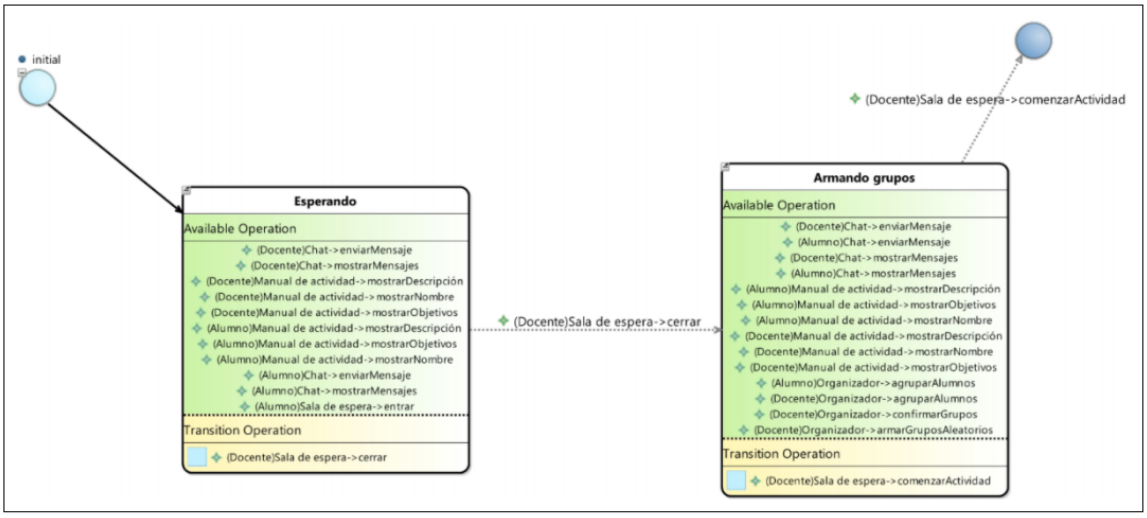

Figura B.4: Actividad “Armando Grupo”.

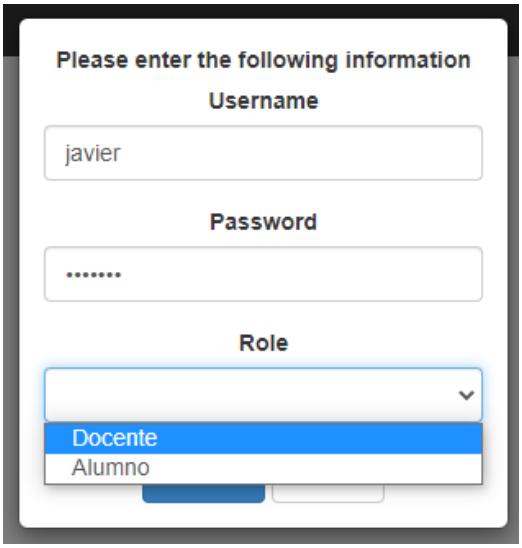

Figura B.5: Login del modelo de clase de robótica.

Figura B.6: Instancia del proceso “Clase de Robótica” con actividad “Selección de Actividad”.

Figura B.7: Ejecución de actividad “Selección de Actividad” para el rol “Docente”.

Figura B.8: Instancias de actividad “ArmandoGrupo”.

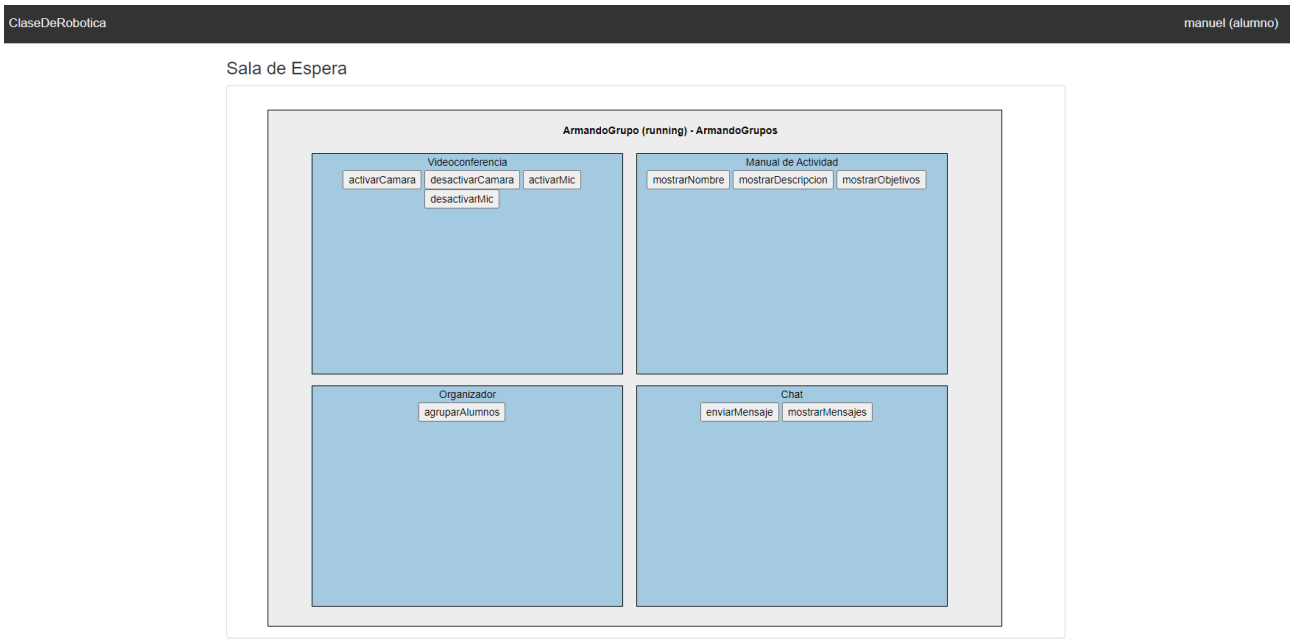

Figura B.9: Ejecución de actividad “ArmandoGrupo” para el rol “Alumno”.

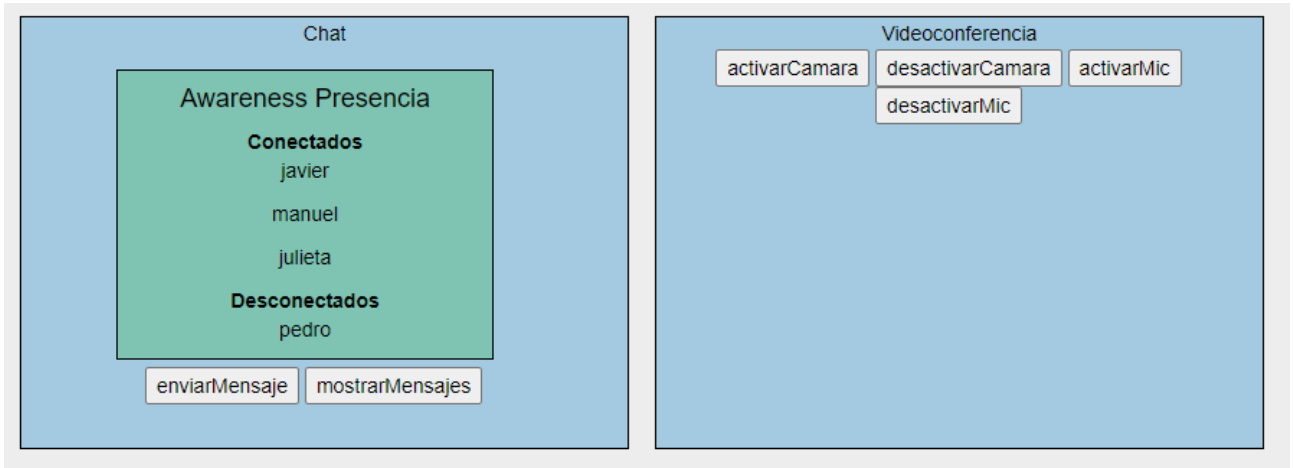

Figura B.10: Actividad con Awareness “Presencia”.

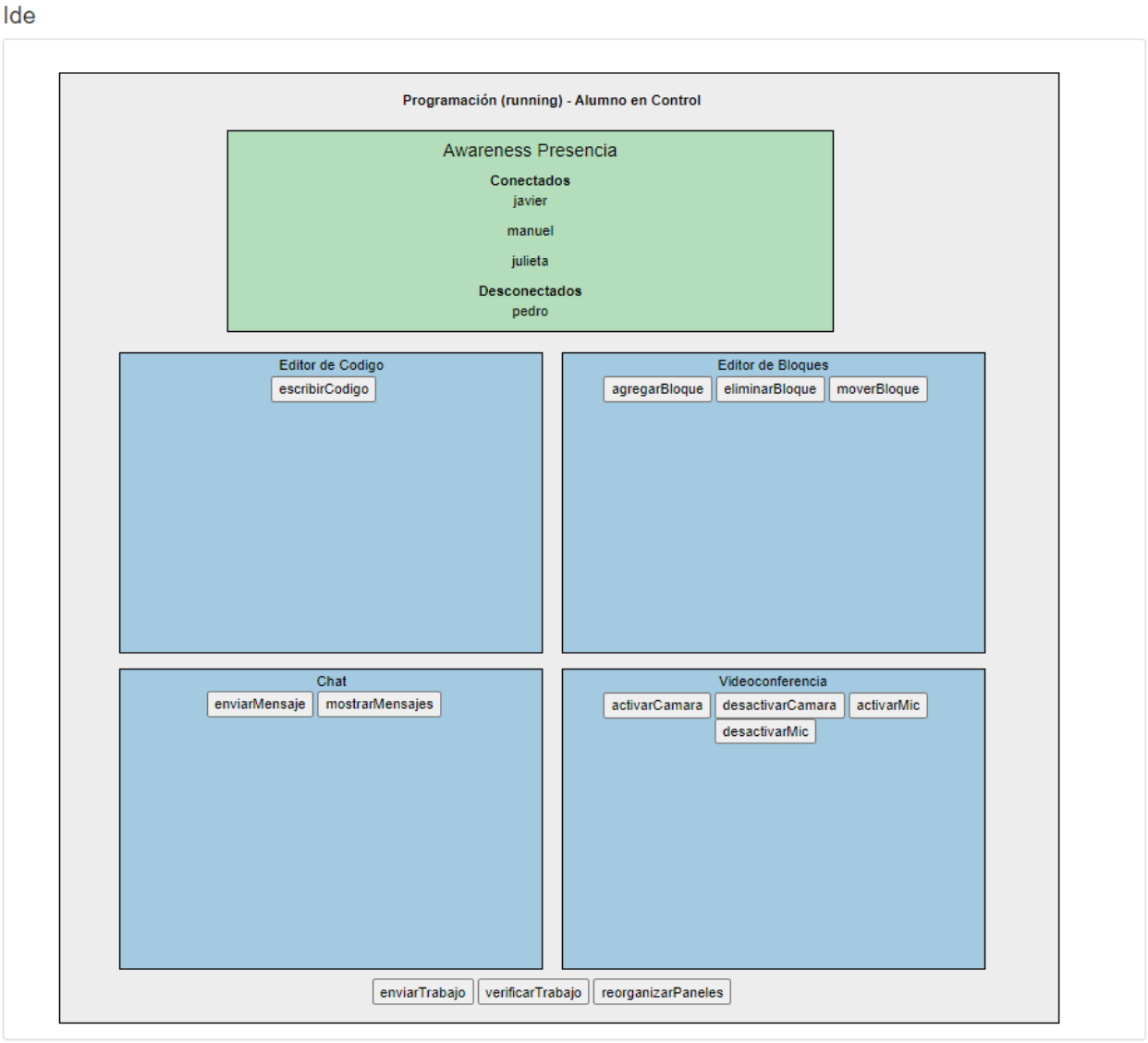

Figura B.11: Ejecución actividad “Programación” con Awareness “Presencia” en Workspace “Ide”.